XDRBG: A Proposed Deterministic Random Bit Generator Based on Any XOF

Anonymous Submission

Abstract. A deterministic random bit generator (DRBG) generates pseudorandom bits from an unpredictable seed, i.e., a seed drawn from any random source with sufficient entropy. The current paper formalizes a security notion for a DRBG, in which an attacker may make any legal sequence of requests to the DRBG and sometimes compromise the DRBG state, but should still not be able to distingush DRBG outputs from ideal random bits. The paper proposes `XDRBG`, a new DRBG based on any eXtendable Output Function (`XOF`) and proves the security of the `XDRBG` in the ideal-`XOF` model. The proven bounds are tight, as demonstrated by matching attacks. The paper also discusses the security of `XDRBG` against quantum attackers. Finally, the paper proposes concrete instantiations of `XDRBG`, employing either the `SHAKE128` or the `SHAKE256` `XDRBG`. Alternative instantiations suitable for lightweight applications can be based on ASCON.

1 Introduction

Generating random bits is a critical function in almost any secure cryptographic system. Usually, the process for generating these random bits is broken into two parts: First, an *entropy source* provides some unpredictable input string as a *seed*. Second, a cryptographic algorithm called a deterministic random bit generator (DRBG) in [3] and this work, and called a PRNG (pseudorandom number generator), cryptographic PRNG, or DRNG (deterministic random number generator) elsewhere, produces the output bits. See [27, 28, 18, 1, 24] for widely-used examples of a DRBG.

Informally, the DRBG must provide output bits which are computationally indistinguishable from ideal random bits, so long as it has been properly seeded. Further, because the internal state of a DRBG might sometimes be compromised, output bits generated before the compromise must remain indistinguishable from ideal random bits, and the DRBG must recover from a state compromise once provided sufficient new entropy. DRBGs are almost always constructed from other cryptographic primitives, such as hash functions, block ciphers, or stream ciphers[1].

In the current paper, we specifically consider a DRBG that follows the interface and requirements of two widely-used standards: SP 800-90A [3] and AIS 20/31 [26, 36].

In [6], *sponge functions* were proposed as a new way of constructing hash functions. These functions use a large fixed permutation or function to process data, but unlike traditional hash functions, they can generate arbitrary-length outputs. The resulting broad class of cryptographic primitive (which might also be realized using other constructions) was named a `XOF` *(eXtended Output Function)* in [32]. Informally, a `XOF` works like a cryptographic hash function, but allows for an arbitrary-length output string.The calls $\texttt{XOF}(x, n)$ and $\texttt{XOF}(x, n + u)$ will yield identical results in their first n bits. A `XOF` can reasonably be modeled as a random oracle with an extremely long output for each query, which is then truncated to the desired output length. See [7] for a security analysis of `XOF`s based on sponge functions such as `SHAKE128` and `SHAKE256`. In recent years, many new sponge-based hash functions

[1] Thus, the DRBGs in [3] are based on AES [29], SHA2 [31], or SHA3 [32], and the DRBG currently used in Linux is based on ChaCha20 [35].

have been proposed; typically these also support XOF functionality. For example, see [37, 2, 5, 14, 15].
In this work, we: (1) Describe XDRBG, a new DRBG based on any XOF. (2) Propose a security notion appropriate for DRBGs. (3) Prove XDRBG secure under this security notion. The proof treats the XOF at hand as a random oracle. (4) Demonstrate classical attacks matching our security bounds. (5) Describe the quantum security of XDRBG. (6) Propose concrete parameters and instantiations for XDRBG. (7) Discuss some remaining open questions.

Relationship with Prior Work. Bertoni et al. describe a generic construction for cryptographic random bit generation based on a sponge construction in [8]. Additional proposals along these lines were made by Gaži and Tessaro[20, 21], Hutchinson [22, 23], and Coretti et al. [11]. While XDRBG is broadly similar to these designs (and has been strongly influenced by the designs of Bertoni et al. and Coretti et al.), our design differs from these earlier works in some important ways:

1. XDRBG supports the interface defined for DRBGs in [3]. Unlike the designs of Bertoni et al. and Coretti et al., XDRBG supports distinct Instantiate and Reseed calls, variable-length outputs from Generate calls, and untrusted additional inputs provided by the caller.
2. XDRBG is intended to be usable with any XOF, with the DRBG making queries to the XOF using a standard interface, and without making any assumptions about its internal workings. Thus, our next DRBG state is part of the output from the XOF instead of remaining in the capacity of the underlying sponge, and is re-input in subsequent XOF queries by the DRBG. Similarly, we model the XOF as an ideal object, rather than considering its underlying structure.
3. XDRBG is targeted for use in the realm of cryptographic random bit generation under SP 800-90 [3, 38, 4] and AIS 20/31 [26, 36], where entropy sources are independently evaluated and validated. We thus feel justified in assuming the availability of known amounts of entropy on demand, with entropy sources that are non-adversarial and whose entropy distributions are oracle-independent. (We thus adopt the model of [16] and later [39], rather than the model of [11].) We suspect that the techniques of [11] could be used to show that the XOF in XDRBG works as a seedless extractor (XDRBG is quite similar to their sponge-based PRNG), but this is left for future work.

2 Preliminaries

2.1 Entropy

A DRBG samples a seed from a random source. The mathematical model for a random source is a distribution, and, for the purpose of the current paper, the all-essential property of a distribution is its min-entropy.
Let $\mathcal{D}^h$ be a distribution over strings $\{0,1\}^*$. We write $S \leftarrow\!\$\, \mathcal{D}^h$ if the string S is chosen according to $\mathcal{D}^h$. In that context, the superscript h indicates a lower bound $h \leq H_{\min}(\mathcal{D}^h)$ for the min-entropy

$$H_{\min}(\mathcal{D}^h) = -\log_2\left(\max_{S \leftarrow\!\$\, \mathcal{D}^h, T\in\{0,1\}^*}(\Pr[S=T])\right),$$

rather than the more traditional Shannon entropy $-\sum_{T\in\{0,1\}^*}(\Pr[S=T]*\log_2(\Pr[S=T]))$. Firstly, the min-entropy of a distribution is always a lower bound for the Shannon entropy of that distribution. Thus, by requiring *at least h bits of min-entropy*, we will always get at least h bits of Shannon entropy. Secondly, in our context the min-entropy is more intuitive: it describes the upper bound for the attacker's chance to guess the seed.
Below, we will consider two thresholds for the min-entropy:

- H_{init}: Whenever the `XDRBG` is instantiated the seed is drawn from a source with at least H_{init} bits of min-entropy.
- H_{rsd}: When the reseed command is called, the seed is drawn from a source with at least H_{rsd} bits of min-entropy.

2.2 Interface for DRBG

Following [3], we define three DRBG operations:

1. $V \leftarrow \textsc{Instantiate}(S, \alpha)$ creates a DRBG state V, using seed material S and a string for optional personalization and other optional data[2] α. The seed S must be drawn from an entropy source with min-entropy H_{init}.
2. $V \leftarrow \textsc{Reseed}(V', S, \alpha)$ creates a DRBG state V from a previous state V', the seed S and the string α. The seed S must be drawn from an entropy source with min-entropy H_{rsd}.
3. $(V, \Sigma) \leftarrow \textsc{Generate}(V', \ell, \alpha)$ generates a new DRBG state V and an ℓ-bit output string Σ from the old state V' and the string α. Σ is required to be indistinguishable from random bits.

In NIST SP 800-90A [3], DRBGs are defined with a particular interface which assumes the DRBG can draw bits from the entropy source directly. Each DRBG has a state of its own, identified with a state handle.
For clarity, we prefer a somewhat simpler interface. Instead of using state handles to keep track of DRBG states, we simply pass in the DRBG state (a bit string in `XDRBG`) to the DRBG function as a parameter. Each DRBG function returns an updated DRBG state. Additionally, when entropy is provided to the DRBG, we draw entropy from the source and pass it in to the DRBG function as a parameter. Note that the cryptographic object being described is unchanged–only the description is different.
At first glance, Instantiate and Reseed may seem redundant, but there is actually an important difference between them. $\textsc{Instantiate}(S, \alpha)$ discards the previous state and the new state only depends on the seed and the optional string α. $\textsc{Reseed}(V, S, \alpha)$ refreshes the state based on the previous state, the seed, and the optional string α. This is reflected by the operations we defined above: Instantiate does not take a DRBG state as input, while Reseed does take a DRBG state as input; both return a resulting DRBG state. This distinction also matters for the security analysis of the DRBG, and the amount of entropy required by each function call, as discussed below.

2.3 Security Level

For typical instantiations of the `XDRBG`, we will discuss their classical and quantum security level, which specify approximately how much computation is needed to distinguish the outputs of the DRBG from random bits. More precisely, if the attacker makes Q `XOF` queries, which implies Q to be a lower bound for the attacker's workload, then we claim $\mathcal{L}$-bit security if the attacker's chance to decide the DRBG output from random is at most $\frac{1}{2} + Q \cdot 2^{-\mathcal{L}}$ for $Q \ll 2^{\mathcal{L}}$, say, $Q \leq 2^{7\mathcal{L}/8}$.
As will become clear below, depending on the instantiation at hand, a classical security level of k bit does not always imply a quantum security level of $k/2$ bit.

2.4 Forward and Backward Security

Informally, forward security (called *backtracking resistance* in [3]) requires that earlier DRBG outputs remain secure when a later DRBG state is compromised. This is a property of the DRBG algorithm. Likewise, backward security (called *prediction resistance* in [3]) requires a DRBG to be able to recover from a compromise of its state. This requires a reseed method (or calling instantiate again) to provide additional

[2] Throughout the paper, we use α (or α_j, $\alpha_{d,i}$, etc.) for those strings.

entropy. A DRBG without access to any new seed material (e.g., from an entropy source) simply cannot achieve backward security.

AIS 20/31 defines two similar properties: *enhanced backward secrecy* and *enhanced forward secrecy*. Enhanced backward secrecy is approximately equivalent to backtracking resistance (aka, forward security); enhanced forward secrecy is approximately equivalent to prediction resistance (aka, backward security).

2.5 Multitarget Attack on INSTANTIATE

Given many devices, each instatiated once, or a few or even one single device instantiated many times, the following multitarget attack becomes possible.

Suppose at each startup the DRBG is instantiated with a seed S containing k bits of entropy. Assume $S \in \{0,1\}^k$ to be uniformly distributed. Assume that after each instantiation at least k bits of output are produced. After I such instantiations, an attacker can recover one DRBG state with a $2^k/I$ search. The attacker simply guesses $2^k/I$ possible values of S, and for each one instantiates the DRBG with S, generates an output, and checks it against the I output values.

This is within the normal security bounds of any k-bit scheme. Nevertheless, it can substantially weaken some applications. For example, consider a device with a 256-bit ECDSA [30] key, supporting 128-bit security. Suppose the device instantiates its DRBG with $k = 128$ random bits at each startup, and that it is restarted and produces an ECDSA signature $I = 2^{32}$ times in its lifetime. In spite of formally supporting 128-bit security, this device is actually vulnerable to a 2^{96}-time attack which will recover its signing key!

Let R be an upper bound on the number of times the DRBG will be instantiated. As long as at least $k + \log_2(R)$ bits of min-entropy is provided for the seed when instantiating the DRBG, the multitarget attack is blocked. As will turn out below, k bits of min-entropy suffice for reseeding.

The requirements for DRBG instantiation in [3] include a *nonce* along with the entropy input. The proposed new requirements for DRBG instantiation in [4] replace the nonce requirement with a requirement for additional entropy. Both the old and new requirements effectively block this multitarget attack in the case that a given user does not instantiate their DRBG more than 2^{64} times.

2.6 Extendable Output Functions (XOFs)

Formally, a `XOF` is a function $\{0,1\}^* \to \{0,1\}^*$, which we model as a random oracle. To avoid returning an infinite sequence, the `XOF` gets an integer as a second parameter[3]: $\texttt{XOF} : \{0,1\}^* \times \mathbf{N} \to \{0,1\}^*$. Now $\texttt{XOF}(x,\ell)$ returns the first ℓ bits from the infinite sequence. Thus, for every $x \in \{0,1\}^*$, the first $\min(\ell,\ell')$ bits of $\texttt{XOF}(x,\ell)$ and $\texttt{XOF}(x,\ell')$ are the same, and, for all $x' \neq x$, the sequences $\texttt{XOF}(x,\ell)$ and $\texttt{XOF}(x',\ell')$ are two independent random sequences of ℓ and ℓ' bits, respectively. The algorithm below describes how our ideal `XOF` might be implemented by lazy sampling.

Of course, any real `XOF` does not provide unlimited ideal security. A typical `XOF` will allow an attacker to make a sequence of queries, and then the attacker will try to differentiate the `XOF` from an ideal `XOF`. Write W for the sum of the lengths of all inputs and outputs (in bit) made by the attacker. We say, a `XOF` supports k-bit security, if for every attacker, the advantage in distinguishing the `XOF` from random is at most $W/2^k$.

FIPS 202 [32] defines two sponge-based `XOF`s: `SHAKE128` and `SHAKE256`. Both employ a cryptographic 1600-bit permutation; `SHAKE128` employs an internal state (the *capacity*) of 256 bits and `SHAKE256` 512 bits.

A sponge-based `XOF` with a capacity of c bits can maintain about $c/2$ bit security

[3] In some applications of a `XOF`, the output length is not known at the time its inputs are provided; these must support a somewhat more complicated interface.

Algorithm 1 `XOF` definition.

function INIT
  $T \leftarrow \{\}$
(# T holds a map $\{0,1\}^* \to \{0,1\}^*$. Initially, T is empty. #)
function `XOF`(x, ℓ)
  if $x \in T$ **then**
    if $|T[x]| < \ell$ **then**
      $s' \leftarrow\$ \{0,1\}^{\ell - |T[x]|}$
      $T[x] \leftarrow T[x] \parallel s'$
  else
    $T[x] \leftarrow\$ \{0,1\}^{\ell}$
(# Now $x \in T$, and $|T[x]| \geq \ell$ #)
  return($T[x]$ truncated to ℓ bits)

against classical attackers [7] and about $c/3$ bit against quantum attackers [13].[4] In that context, the term *security* has to be understood as *indifferentiability from a random oracle*, when the underlying cryptographic permutation is modelled as a random permutation.
Accordingly, `SHAKE128` can claim 128 bit security against classical and 85 bit security against quantum attackers, and `SHAKE256` can claim 256-bit security against classical and 171 bit security against quantum attackers.

3 `XDRBG` Definition

`XDRBG` is a DRBG based on an underlying `XOF`. One could also view `XDRBG` as a *mode of operation* for a `XOF` to realize a DRBG. Note that unlike most prior designs of this kind, `XDRBG` does not assume anything about the internal structure of the `XOF`.

Conventions and Notation.

1. When we encode an integer as a bitstring, we always assume network byte order, and write X_n to represent encoding X as an n-bit integer.
2. For each instantiation of the `XDRBG` we will claim two approximate security levels: one with respect to classical attackers, and a second one with respect to quantum attackers employing Grover's algorithm. A security level of k bit implies that the given attacker, when restricted to time $t < 2^k$, succeeds with at most $t/2^k$ probability.
   The classical (quantum) security level of an instantiation of the `XDRBG`, employing a given `XOF`, is the minimum of the classical (quantum) security level of the `XDRBG` in the ideal-`XOF` model and the classical (quantum) security level of the `XOF` at hand.
3. The internal state of `XDRBG` is a single bitstring, V of fixed size $|V|$.
4. All `XDRBG` functions support an additional input α to personalize the DRBG; α can be empty. We assume a function ENCODE, such that the string α and the number $n \in \{0,1,2\}$ can be uniquely recovered from $\text{ENCODE}(S, \alpha, n)$ for any string S. In other words, there must be no $(S, \alpha, n) \neq (S', \alpha', n')$ with $\text{ENCODE}(S, \alpha, n) = \text{ENCODE}(S', \alpha', n')$. We write $|\text{ENCODE}| = |\text{ENCODE}(S, \alpha, n)| - |S| - |\alpha|$ for the stretch of the encoding. See appendix B for example encodings.
   Within our proofs, we also assume a function PARSE. If S can be written as $S = \text{ENCODE}(s_1, s_2, i)$, the function returns $\text{PARSE}(S, 1) = s_1$, $\text{PARSE}(S, 2) = s_2$,

[4] Cautionary note: To the best of our knowledge, the claimed $c/3$ bits of quantum security for a sponge with c bit capacity has so far only been published at an eprint server [13], but not yet at a peer-reviewed conference or journal. In this paper, we assume the claim to be correct.

and PARSE$(S, 3) = i$. Further, we use the convention that PARSE$(S, -1)$ gives the last entry in the tuple that was ENCODEd. When there is no tth element in the tuple that was encoded to construct the string S, PARSE(S, t) returns a failure symbol $\perp$.

5. Output lengths are specified in bits. As discussed in Section 7, we recommend an upper limit on the output from each GENERATE call, called `maxout`.

6. INSTANTIATE and RESEED calls require entropy to result in a secure DRBG state. We thus define two parameters specifying how much min-entropy must be provided. Each INSTANTIATE requires at least H_{init} bits of min-entropy; each RESEED requires at least H_{rsd} bits of min-entropy.

The security analysis of `XDRBG` can continue without defining the size of the state ($|V|$) or the entropy bounds H_{rsd} and H_{init}. Concrete recommendations for these parameters – and a possible `maxout` limit for generate requests – appear in Section 7.

Algorithm 2 `XDRBG` Definition

The function ENCODE : $\{0,1\}^* \times \{0,1\}^* \times \{0,1,2\} \to \{0,1\}^*$ takes two strings and a number and returns a string. We require ENCODE$(s_1, s_2, i) \neq$ ENCODE(s_1', s_2', i') for $(s_1, s_2, i) \neq (s_1', s_2', i') \in \{0,1\}^* \times \{0,1\}^* \times \{0,1,2\}$.

function INSTANTIATE(seed, α)
  (# Returns $|V|$-bit state; source for seed: $\geq H_{init}$ bit min-entropy. #)
  $V \leftarrow$ `XOF`(ENCODE(seed, α, 0), $|V|$)
  return(V)

function RESEED(V', seed, α)
  (# Returns $|V|$-bit state; source for seed: $\geq H_{rsd}$ bit min-entropy. #)
  $V \leftarrow$ `XOF`(ENCODE(($V' \parallel$ seed), α, 1), $|V|$)
  return(V)

function GENERATE(V', ℓ, α)
  (# Returns $|V|$-bit state and ℓ-bit string Σ. #)
  $T \leftarrow$ `XOF`(ENCODE($V', \alpha, 2$), $\ell + |V|$)
  $V \leftarrow$ first $|V|$ bits of T
  $\Sigma \leftarrow$ last ℓ bits of T
  return(V, Σ)

Design Rationale. The goal of `XDRBG` is to provide an efficient and comprehensible DRBG based on any `XOF`. A simple, comprehensible design makes implementation, cryptanalysis, proving security, and checking the proof all easier. These considerations led us to define `XDRBG` so that: (1) Each call to a DRBG function (INSTANTIATE, RESEED, or GENERATE) results in a single `XOF` query, and (2) The first $|V|$ bits of the `XOF` output always become the new state V. (In GENERATE calls, the remaining bits of the `XOF` output become the DRBG output.)

4 The DRBG Security Game

In order to reason about the security of our DRBG, we first need to define what it means for a DRBG to be secure. Informally, our security goals can be summarized as follows: *The attacker must not be able to distinguish the outputs from the DRBG from perfect random bits.* This must be true for any sequence of instantiate, generate and reseed calls to different devices. The only exception is output generated in the time following a state compromise and before the next intake of fresh entropy (i.e., before either reseeding or instantiating the DRBG).

For clarity of explanation, we will use the following terms in discussing the game:

- When the challenger or attacker interact with the `XOF`, this is a *query*. (Example: The challenger makes a `XOF` query.)
- When the challenger interacts with a DRBG instance, this is a *call*. (Example: The challenger makes an INSTANTIATE call.)
- When the attacker interacts with the challenger, this is a *request*. (Example: The attacker makes a `R_OUT` request.)

Thus, when the attacker makes a `R_OUT` request, this causes the challenger to make a GENERATE call to the DRBG, which then causes the challenger to make a `XOF` query.

4.1 Intuition for the Game

Broadly spoken, the goal of this security game is to capture all the ways an attacker might interact with devices implementing SP 800-90A type DRBGs. The attacker requests some devices to generate outputs, to reseed, or to instantiate. The challenger simulates all devices using the DRBG and reacts on the attacker's requests. The only constraint is that the first request to each device must be instantiate. The attacker is free to choose the *additional input* (α) for the requests. After every request, the attacker can learn the DRBG state of a device by *compromising* it. The device must recover from a compromise when an instantiate or reseed call is made.
The attacker can force each device to repeatedly be instantiated. This captures the multitarget instantiation attack on a single device described above, as well as any other weaknesses in the instantiation process the DRBG might suffer from. By allowing the attacker to repeatedly compromise the state of the device and to reseed it, we capture any flaws in either the reseed process or in the backtracking or prediction resistance of the DRBG.
The attacker's goal is to distinguish the DRBG output from independent uniformly distributed random bits. But whenever a device state is compromised, i.e., has become known to the attacker, the attacker can compute the device's DRBG outputs on its own, and thus trivially distinguish them from random bits. We thus consider the output bits generated after a compromise and before the next reseed or instantiate as corrupted and exclude them from the distinguishing goal.

4.2 Game Definition and Rationale

To capture the above intuition formally, we specify a security game, see algorithm 3. It can be seen as an extension of the proof model used in [39]. Our game incorporates a wider range of possible attacks, and closely tracks with the assumptions and requirements of the SP 800-90 series and AIS 20/31:

1. At the beginning of the game, the challenger generates a random bit b. If $b = 0$, all `R_OUT` requests will be answered by outputs from the DRBG; if $b = 1$, some of those requests will be answered with ideal random bits, instead. All other requests are answered in exactly the same way regardless of b.
2. We assume D devices the adversary can make requests to. The sum of all requests sent to all devices is exactly R. (An adversary making $R' < R$ requests can always be modelled by an adversary making exactly R requests while ignoring any responses after the first R' requests.)
3. An adversarial request is a quintuple $(\texttt{req}, d, i, \alpha_{d,i}, \texttt{leak})$:
   - $\texttt{req} \in \{\texttt{R\_INST}, \texttt{R\_RESEED}, \texttt{R\_OUT}(\ell)\}$ defines the call the device needs to make to handle the request.
   - $d \in \{1, \ldots, D\}$ is the index of the device the request goes to.
   - i is a counter: $(\texttt{req}, d, i, \alpha_{d,i}, \texttt{leak})$ is the i-th request to device d.
   - $\alpha_{d,i}$ denotes the additional information for the call.
   - $\texttt{leak} \in \{\mathsf{true}, \mathsf{false}\}$ indicates a state compromise: if true, the adversary learns $V_{d,i}$, i.e., the state of device d at the end of handling the request.

Algorithm 3 DRBG Security Game:
The attacker can access up to D devices and makes R requests in total. The challenger responds to valid requests. The attacker can directly query the `XOF` up to Q times, not counting the challenger's own `XOF` queries, made when addressing the R requests. The attacker wins if the final message from the challenger is 1.

function CHALLENGER(D, Q, R)
(# Start the game: randomly choose the secret bit b. #)
  $b \leftarrow\$ \{0,1\}$
(# Initiate tracker for previous requests #)
  for $D \in \{1,\ldots,D\}, i \in \{1,\ldots R\}$ **do**
    $\text{Done}(d,i) = \mathsf{false}$
(# Attacker first commits to distributions and then is granted access to `XOF`*. #)*
  for $i \in \{1,\ldots,U\}, j \in \{1,\ldots R\}, h \in \{H_{\text{init}}, H_{\text{rsd}}\}$ **do**
    Attacker chooses distributions $\mathcal{D}^h_{i,j}$, as elaborated in the text.
  Challenger grants direct access to `XOF` for attacker, for up to Q queries.
(# Attacker makes R requests. Handle valid requests, ignore invalid ones. #)
  for step $\in \{1,\ldots,R\}$ **do**
    Attacker chooses request $(\texttt{req}_{d,i}, d, i, \alpha_{d,i}, \texttt{leak})$, as elaborated in the text.
    if $((\overline{\text{Done}(d,i)}) \wedge (i=1) \wedge (\texttt{req}_{d,i} = \texttt{R\_INST})) \vee ((i>1) \wedge \text{Done}(d,i-1))$ **then**
      $V_{d,i} \leftarrow$ Handle_Request$(b, \texttt{req}_{d,i}, d, i, \alpha_{d,i}, \mathcal{D}^{H_{\text{init}}}_{d,i}, \mathcal{D}^{H_{\text{rsd}}}_{d,i})$ *(# see Alg. 4 #)*
      $\text{Done}(d,i) \leftarrow \mathsf{true}$
      if `leak` **then** *(# Compromise current state of device d. #)*
        Send DRBG state $V_{d,i}$ to attacker.
        $\texttt{corrupt}_d \leftarrow \mathsf{true}$
(# Finish the game: the attacker wins if it correctly guesses the secret bit b. #)
  Receive $\hat{b}$ from attacker.
  if $b = \hat{b}$ **then** send 1 to attacker **else** send 0 to attacker.

Algorithm 4 Subroutine Handle_Request from DRBG Security Game.

function HANDLE_REQUEST($b, \texttt{req}_{d,i}, d, i, \alpha_{d,i}, \mathcal{D}^{H_{\text{init}}}_{d,i}, \mathcal{D}^{H_{\text{rsd}}}_{d,i}$)
  if req is `R_OUT`(ℓ) **then**
    $(V_{d,i}, Z) \leftarrow$ GENERATE$(V_{d,i-1}, \ell, \alpha_{d,i})$
    if $b = 1$ AND NOT $\texttt{corrupt}_d$ **then**
      $Z \leftarrow\$ \{0,1\}^n$
    Send Z to attacker.
  else if req is `R_RESEED` **then**
    $S_{d,i} \leftarrow\$ \mathcal{D}^{H_{\text{rsd}}}_{d,i}$
    $V_{d,i} \leftarrow$ RESEED$(V_{d,i-1}, S_{d,i}, \alpha_{d,i})$
    $\texttt{corrupt}_d \leftarrow \mathsf{false}$
  else if req is `R_INST` **then**
    $S_{d,i} \leftarrow\$ \mathcal{D}^{H_{\text{init}}}_{d,i}$
    $V_{d,i} \leftarrow$ INSTANTIATE$(S_{d,i}, \alpha_{d,i})$
    $\texttt{corrupt}_d \leftarrow \mathsf{false}$
  return $(V_{d,i})$

4. A request $(\texttt{req}, d, i, \alpha_{d,i}, \texttt{leak})$ is *valid*, if and only if
   - there has been no i-th valid request to device d before, and
   - either this is the first valid request to device d (i.e., $i = 1$) and the device is requested to instantiate ($\texttt{req} = \texttt{R\_INST}$) or this is not the first valid request to device d (i.e., $i > 1$) and the $(i-1)$-st valid request to device d has already been handled.

   The challenger only reacts to valid requests, cf. lines 10–15 from algorithm 3. Note the usage of the array to keep track of valid requests.

   This serves two purposes: It prevents requests to uninstantiated devices. And it enforces a consistent notation: Requests to device d are numbered by $i = 1$, $i = 2$, $i = 3$, ..., in that order and without skipping or repeating an integer.

5. Each valid request results in a single DRBG call made by the challenger (cf. Alg. 4). Each DRBG call leads to one single `XOF` query (cf. Alg. 2). In parallel to the requests, the attacker can also query the `XOF` up to Q times (cf. Line 7 from Alg. 3).

6. For each device, we assume the availability of a properly designed and tested entropy source (known as an NTG.1, PTG.2 or PTG.3 in AIS 20/31) with a specified min-entropy. As pointed out above, we assume lower bounds H_{init} and H_{rsd} for the min-entropy of our entropy sources, namely H_{init} when the DRBG is instantiated, and H_{rsd} when it is reseeded.

   SP 800-90C requires a security parameter k and fixes the entropy bounds by $H_{\text{init}} = 3k/2$, and $H_{\text{rsd}} = k$. AIS 20/31 requires[5] $H_{\text{init}} = H_{\text{rsd}} = 240$.

   Assuming the specified min-entropy, we claim the validity of our results for *all realistic* entropy sources.

   - At the beginning of the game, the attacker will specify a sequence of distributions $\mathcal{D}^h_{d,1}, \ldots, \mathcal{D}^h_{d,R}$ with min-entropy $h \in \{H_{\text{rsd}}, H_{\text{init}}\}$ for each device d. As the distributions are formally specified by the attacker, they cover *all* realistic entropy sources with the required min-entropy.
     Obviously, the entropy sources from different devices can be distributed differently. Furthermore, a realistic entropy source may change its distribution over time, e.g., due to heating up or cooling down. Thus, the attacker must choose a distribution $\mathcal{D}^h_{d,i}$ for each possible triple (d, i, h).
   - To restrict the entropy sources to *realistic* ones, we require the attacker to commit to all distributions $\mathcal{D}^h_{d,i}$ at the very beginning of the game, in advance of all queries to the `XOF`, either directly by the attacker querying the `XOF`, or indirectly from the attacker's requests (cf. lines 5–7 of algorithm 3).[6] This will allow us to apply Lemma 1 below: one cannot choose $u \neq u'$ with $\Pr[\texttt{XOF}(u, \ell) = \texttt{XOF}(u', \ell)] > 2^{-\ell}$, without first querying the `XOF`.

7. In some situations, the attacker may realistically be able to *compromise* a device i.e., to learn its internal state $V_{d,i}$. This is indicated by `leak`, see above.

   For each device d, the algorithm maintains a flag $\texttt{corrupt}_d$. The flag is set when the device's state is compromised, and the flag is cleared when the state is advanced randomly using fresh entropy, i.e., after each reseed and instantiate request. (There is no need to initialize $\texttt{corrupt}_d$, because the first valid request to any device d is always instantiate, which sets $\texttt{corrupt}_d$ without prior reading.)

[5] We disregard the alternative of requiring 250 bit of Shannon Entropy, in AIS 20/31. Also, AIS 20/31 imposes some additional requirements on seeding a DRNG [36], which we also disregard here.

[6] In reality, the distributions stem from random sources with the required amount of entropy. Giving the attacker the ability to choose those distributions $\mathcal{D}^h_{d,i}$ is not realistic, but matches the all-quantification: I.e., if we claim (as we actually do) security for all combinations of distributions with the required min-entropy, we can, as well, model this by the adversary choosing the combination of distributions it likes. On the other hand, if the attacker had access to the `XOF` before committing to $\mathcal{D}^h_i$, it might choose some $\mathcal{D}^h_i$ with $\Pr[\texttt{XOF}(u, \ell) = \texttt{XOF}(u', \ell)] > 2^{-h} + 2^{-\ell}$ for $\ell \geq 1$ and $u, u' \leftarrow\!\!\$\, \mathcal{D}^h_i$, even though $\Pr[u = u'] \leq 2^{-h}$. This is not a realistic attack setting for a realistic entropy source.

A stronger attack model might allow the attacker to *set* the device's state to a chosen or known value. Our game does not support this, because we consider such attacks unrealistic. But we briefly discuss their potential impact in section 8.3.

8. At the end of the game, the attacker tries to guess the random bit b chosen at the beginning of the game.

   In order for XDRBG to be acceptably secure, the probability of the attacker to win the game must be no greater than $1/2 + \epsilon$ for some very small ϵ.

5 Security Analysis

5.1 The Main Result and some Corollaries

Our results depend on the maximum number λ_1 of queries (d, i) with the same $\alpha_{d,i}$ and and on the number λ_2 of unordered pairs $(d, i) \neq (d', i')$ with $\alpha_{d,i} = \alpha_{d',i'}$:

$$\lambda_1 = \max_a \left(\left| \{(d,i) : \alpha_{d,i} = a\} \right| \right) \quad \text{and} \quad \lambda_2 = \sum_a \left(\binom{\left| \{(d,i) : \alpha_{d,i} = a\} \right|}{2} \right). \tag{1}$$

We assume $\binom{0}{2} = 0 = \binom{1}{2}$, and in general $\binom{N}{2} = \frac{N(N-1)}{2}$.

Theorem 1. *Let H_{init} and H_{rsd} be the min-entropy for* R_INST *and* R_RESEED *requests, respectively. Let $|V| \geq H_{init}$ be the state size of the DRBG. Let the attacker make Q queries and R requests and let λ_1 and λ_2 be defined as in Equation 1.*
The attacker's probability to win the DRBG game is at most $1/2 + \epsilon$, with

$$\epsilon \leq Q \left(\frac{\lambda_1}{2^{H_{init}} - Q} + \frac{1}{2^{H_{rsd}} - Q} \right) + \frac{\lambda_2}{2^{H_{init}}} + \frac{R^2}{2 \cdot 2^{|V|}} \tag{2}$$

Below, we describe the security we guarantee, depending on different policies for the choice of the $\alpha_{d,i}$. We assume $\log_2(Q) \ll H_{\text{rsd}} \leq H_{\text{init}} \leq |V|$.

The first policy imposes *no restrictions* on the choice of the $\alpha_{d,i}$. The $\alpha_{d,i}$ may even be empty, or set to another constant string. With at most R adversarial requests, $\lambda_1 \leq R$ and $\lambda_2 \leq \binom{R}{2} \leq \frac{R^2}{2}$, which gives the following bound:

Corollary 1. *Let H_{init} and H_{rsd} be the min-entropy for* R_INST *and* R_RESEED *requests, respectively. Let $|V| \geq H_{init}$ be the state size of the DRBG.*
The attacker's probability to win the DRBG game, making Q queries and R requests, is at most $1/2 + \epsilon$, with

$$\epsilon \leq Q \cdot \left(\frac{R}{2^{H_{init}} - Q} + \frac{1}{2^{H_{rsd}} - Q} \right) + \frac{R^2}{2} \cdot \left(\frac{1}{2^{H_{init}}} + \frac{1}{2^{|V|}} \right).$$

The above corollary incentivizes to choose $H_{\text{rsd}} < H_{\text{init}}$, actually: $H_{\text{rsd}} \approx H_{\text{init}} - \log_2(R)$. However, we get inproved bounds by requiring disjoint α. The second policy thus requires *unique* $\alpha_{d,i}$, i.e., if $\alpha_{d,i} \neq \alpha_{d',i'}$ for $(d, i) \neq (d', i')$. In this case, we have $\lambda_1 = 1$ and $\lambda_2 = 0$ and an improved bound:

Corollary 2. *Let H_{init} and H_{rsd} be the min-entropy for* R_INST *and* R_RESEED *requests, respectively. Let $|V| \geq H_{init}$ be the state size of the DRBG. Let all additional inputs $\alpha_{d,i}$ be unique. Let the attacker make Q queries and R requests.*
The attacker's probability to win the DRBG game is at most $1/2 + \epsilon$, with

$$\epsilon \leq Q \cdot \left(\frac{1}{2^{H_{init}} - Q} + \frac{1}{2^{H_{rsd}} - Q} \right) + \frac{R^2}{2 \cdot 2^{|V|}}.$$

I.e., *unique* additional data $\alpha_{d,i}$ would incentivise $H_{\text{init}} = H_{\text{rsd}}$. On the other hand, the $\alpha_{d,i}$ are de-facto nonces, and handling nonces may not always be desirable for a DRBG. Actually, the current draft of SP800-90 [4] abandons the requirement to use a nonce, which has been imposed so far [3]. Instead, [4] explicitly requires $H_{\text{init}} > H_{\text{rsd}}$. So finally, we study a policy *in between* no rules and strict uniqueness for all $\alpha_{d,i}$. Consider D devices using the same `XDRBG` with the same `XOF` in total. Each device d has a unique name id_d, i.e., $\text{id}_d \neq \text{id}_{d'}$ for $d \neq d'$. The adversary makes at most R_1 requests to each device (thus, the total number of requests is at most $R = D * R_1$). Now set $\alpha_{d,i} = \text{id}_d$, i.e., all requests to a device just use the device's unique name as the additional input. Then $\lambda_1 \leq R_1$ and $\lambda_2 \leq D * \binom{R_1}{2} \leq \frac{R_1^2}{2D}$, and thus:

Corollary 3. *Let H_{init} and H_{rsd} be the min-entropy for* `R_INST` *and* `R_RESEED` *requests, respectively. Let $|V| \geq H_{init}$ be the state size of the DRBG. Let id_d be a unique name for each device d and $\alpha_{d,i} = id_d$. Let there be D devices, and each device will not respond to more than R_1 requests. Set $R = D * R_1$ and let the attacker make Q queries.*
The attacker's probability to win the DRBG game is at most $1/2 + \epsilon$, with

$$\epsilon \leq Q \cdot \left(\frac{R_1}{2^{H_{init}}} + \frac{1}{2^{H_{rsd}}}\right) + \frac{R^2}{2D} \times \left(\frac{1}{2^{H_{init}}} + \frac{1}{2^{|V|}}\right). \tag{3}$$

Once again, this incentivizes $H_{\text{rsd}} < H_{\text{init}}$, but now $H_{\text{rsd}} \approx H_{\text{init}} - \log_2(R_1)$.

Remark. Sometimes, devices $d \neq d'$ with identical names: $\text{id}_d = \text{id}_{d'}$ may exist. E.g., device names may be chosen at random. In this case, the claimed security bound still holds if we treat all devices with the same name as a single device. Namely, if we redefine R_1 such that all devices d_1, d_2, ... with $\text{id}_{d_1} = \text{id}_{d_2} = \cdots$ together do not respond to more than R_1 queries, then equation 3 still applies.

5.2 The Proof of the Main Result

Notation for the Proof. As a shorthand for $(b, \texttt{req}_{d,i}, d, i, \alpha_{d,i}, \texttt{leak})$ we use (d, i). A generate request (d, i) is *corrupted*, if, when Handle_Request is called, $\texttt{corrupt}_d = \texttt{true}$. Else it is *faithful*. Instantiate and reseed requests are always faithful[7]
We refer to the inputs to the attacker's `XOF` queries as $W_1, \ldots, W_Q$. Furthermore, recall that $V_{d,i}$ denotes the i-th state on device d. We write $U_{d,i}$ for the matching input from the challenger's matching `XOF` query, i.e., $V_{d,i} = \texttt{XOF}(U_{d,i}, |V|)$:

- $U_{d,i} = \text{ENCODE}(S_{d,i}, \alpha_{d,i}, 0)$ for instantiate,
- $U_{d,i} = \text{ENCODE}((V_{d,i-1} \parallel S_{d,i}), \alpha_{d,i}, 1)$ for reseed, and
- $U_{d,i} = \text{ENCODE}(V_{d,i-1}, \alpha_{d,i}, 2)$ for generate.

Also, we say U, d, i is *corrupted* or *faithful*, if (d, i) is so.
If W_j is a `XOF` query made by the attacker, write Q_n for the number of queries in $W_1, \ldots, W_Q$ with $\text{PARSE}(W_j, 3) = n$. Clearly, $Q_1 + Q_2 + Q_3 \leq Q$.

Independent `XOF` queries. Our proof consists of a sequence of lemmas. The first one is almost trivial.

Lemma 1. *For any $u \neq u'$ chosen independently from the* `XOF` *and any $\ell \geq 1$*

$$\Pr[\texttt{XOF}(u, \ell) = \texttt{XOF}(u', \ell)] = 2^{-\ell}.$$

Proof. Recall algorithm 1. Since $u \neq u'$, the values $\texttt{XOF}(u, \ell)$ and $\texttt{XOF}(u', \ell)$ are chosen as two independent random ℓ-bit values. Their probability to collide is $2^{-\ell}$. □

[7] Note that the security game *first* calls Handle_Request and *only then* considers the `leak` parameter. Thus, a request $(d, i) = (b, \texttt{req}_{d,i}, d, i, \alpha_{d,i}, \texttt{leak})$ can be corrupted or faithful, regardless of `leak`. But if $\texttt{leak} = \texttt{true}$ then all subsequent generate requests $(d, i+1)$, $(d, i+2)$, ... to the same device will be corrupted, until the first reseed or instantiate request is made.

Collisions. A part of our proof is based on bounding the probability of an input W_j for a XOF query made by the attacker to collide with a faithful $U_{d,i}$, i.e., the event $W_j = U_{d,i}$. We refer to the triple (j, d, i) as a *collision*. A collision can only occur if the input strings to the XOF are identical, and thus for every t, $\text{PARSE}(W_j, t) = \text{PARSE}(U_{d,i}, t)$. Specifically, this requires that $\text{PARSE}(W_j, -1) = \text{PARSE}(U_{d,i}, -1) = \alpha_{d,i}$, so that a given attacker XOF query can only collide with a XOF query from one kind of XDRBG call. Note that for any given string a, there are at most λ_1 requests (d, i) with $\alpha_{d,i} = a$.

Instantiate Collisions. We start with collisions (j, d, i), where (d, i) is an instantiate request, i.e., $\text{PARSE}(W_i, -1) = \text{PARSE}(U_{d,i}, -1) = 0$.

Lemma 2. *Let BAD_{UW}^{INIT} denote the event that during the attack game a XOF query j and an instantiate request (d, i) is made with $U_{d,i} = W_j$. Then*

$$\Pr[BAD_{UW}^{INIT}] \leq \frac{Q_1 \cdot \lambda_1}{2^{H_{init}} - Q}.$$

Proof. An instantiate collision (j, d, i) implies that the attacker actually did make both the XOF query W_j and the instantiate request (d, i) during the DRBG security game. If the query before request j, the attacker is trying to guess a random value with H_{init} bit of min-entropy, which will be chosen later (i.e., when the request is made). In this case, $\Pr[U_{d,i} = W_j] \leq 2^{-H_{\text{init}}}$. If request j is made before the query, and the event $\text{BAD}_{\text{UW}}^{\text{INIT}}$ did not already occur before the request, then the attacker is trying to guess a fixed unknown value with H_{init} bit min-entropy. But in this case, the attacker may already know at most $j - 1$ relationships $W_1 \neq U_{d,i}, \ldots, W_{j-1} \neq U_{d,i}$. Thus, the attacker's chance to guess $U_{d,i}$ when making up to Q_1 queries W_j with $\text{PARSE}(W_j, -1) = 1$ is $\Pr[U_{d,i} = W_j] \leq 1/(2^{H_{\text{init}}} - Q_1)$.
As there are at most $Q_1 \cdot \lambda_1$ triples (j, d, i), with (d, i) being an instantiate query and $\alpha_{d,i} = \text{PARSE}(W_j, 2)$, the probability of any instantiate collision (j, d, i) is

$$\Pr[\text{BAD}_{\text{UW}}^{\text{INIT}}] \leq \frac{Q_1 \cdot \lambda_1}{2^{H_{\text{init}}} - Q_1} \leq \frac{Q_1 \cdot \lambda_1}{2^{H_{\text{init}}} - Q}.$$

□

Repeating States. Another event, which can be beneficial for the attacker, is the case that the challenger generates the same state $V_{d,i} = V_{d',i'}$ from two different requests (d, i) and (d', i').

Lemma 3. *Let BAD_{UW}^{INIT} be defined as in lemma 2. Let BAD_{VV} denote the event that some of the challenger's states during the attack game repeat. I.e., for some $(d, i) \neq (d', i')$ it holds that $V_{d,i} = V_{d',i'}$. Then*

$$\Pr[\text{BAD}_{\text{VV}} | \overline{BAD_{UW}^{INIT}}] \leq \frac{\lambda_2}{2^{H_{init}}} + \frac{R^2}{2 \cdot 2^{|V|}}$$

Proof. Consider two requests (d, i) and (d', i'), where (d, i) is made before (d', i'). If $i > 1 < j$ then assume $V_{d,i-1} \neq V_{d',j'-1}$ (otherwise, the event BAD_{VV} has already occurred). If $U_{d,i} \neq U_{d',i'}$ then

$$\Pr[V_{d,i} = V_{d',i'} | U_{d,i} \neq U_{d',i'}] = 1/2^{|V|},$$

cf. lemma 1. As there exist at most $\binom{R}{2}$ unordered pairs $U_{d,i} \neq U_{d',i'}$

$$\Pr[\exists d, i, d', i' : (U_{d,i} \neq U_{d',i'} \wedge V_{d,i} = V_{d',i'})] \leq \binom{R}{2} \frac{1}{|V|} \leq \frac{R^2}{2 \cdot 2^{|V|}}. \tag{4}$$

We still need to discuss $\Pr[U_{d,i} = U_{d',i'}]$. If at least one of the requests (d, i), (d', i') is not an instantiate request, then then $U_{d,i} \neq U_{d',i'}$ follows from $V_{d,i-1} \neq V_{d',j'-1}$. If $\alpha_{d,i} \neq \alpha_{d',i'}$, then $U_{d,i} \neq U_{d',i'}$.

To bound $\Pr[U_{d,i} = U_{d',i'}]$, we can thus assume ($\alpha_{d,i} = \alpha_{d',i'}$) and both (d,i) and (d',i') are instantiate requests. As the seeds $S_{d,i}$ and $S_{d',i'}$ are independently drawn, and the min-entropy of each seed is at most H_{init}, we get

$$\Pr[U_{d,i} = U_{d',i'}] \leq \Pr[S_{d,i} = S_{d',i'}] \leq \frac{1}{2^{H_{\text{init}}}}.$$

As there exist at most λ_2 pairs $(d,i)/neq(d',i)$ with $ai_{d,i} = \alpha_{d',i'}$, we thus have

$$\Pr[\exists d,i,d',i' : U_{d,i} = U_{d',i'} \leq \frac{\lambda_2}{2^{H_{\text{init}}}}. \tag{5}$$

The lemma follows from combining equations 4 and 5. □

Reseed Collisions. How likely is a collision (j,d,i), if (d,i) is a reseed request?

Lemma 4. *Let BAD_{VV} be as defined in lemma 3. Let BAD^{RSD}_{UW} denote the event that (j,d,i) exist, where (d,j) is a reseed query and $U_{d,i} = W_j$. Then*

$$\Pr[BAD^{RSD}_{UW} | \overline{BAD_{VV}}] \leq \frac{Q_2}{2^{H_{rsd}} - Q}.$$

Proof. We start with bounding the probability of $\text{BAD}^{\text{RSD}}_{\text{UW}}$. Assume (d,i) to be a reseed request, and $U_{d,i} \notin \{W_1, \ldots, W_{j-1}\}$. Consider the probability of the attacker to choose W_j with $W_j = U_{d,i}$. Even if the attacker knows $V_{d,j-1}$, maybe due to a previous compromise, it still has to guess the current seed $S_{d,i}$. The min-entropy of $S_{d,i}$ is H_{rsd} bits.
Assume that, up to this point, $\text{BAD}^{\text{RSD}}_{\text{UW}}$ did not yet happen. This implies $U_{d,i} \notin \{W_1, \ldots, W_j\}$, but otherwise, the attacker has no knowledge about the seed $S_{d,i}$. Thus, the probability to guess $U_{d,i}$ when choosing W_j with $\text{PARSE}(W_j, -1) = 2$ is $\Pr[W_j = U_{d,i}] \leq \frac{1}{2^{H_{\text{rsd}}} - Q_2}$.
Note that, even if the attacker knows $V_{d,i-1}$, we assume no repeating states (i.e., $\overline{\text{BAD}_{\text{VV}}}$), so for any given W_j, there can be at most one request (d,i) such that (j,d,i) could possibly collide, namely the unique (d,i) with $\text{PARSE}(W_j, 1) = V_{d,i-1}$. So in total, the attacker's chance to find a collision within Q queries is

$$\Pr[\text{BAD}^{\text{RSD}}_{\text{UW}} | \bar{[}\text{BAD}_{\text{VV}}] \leq \frac{Q_2}{2^{H_{\text{rsd}}} - Q_2} \leq \frac{Q_2}{2^{H_{\text{rsd}}} - Q}$$

□

Faithful Generate Requests. In the case of a generate request, observe that if (d,i) is compromised, i.e., if the attacker knows $V_{d,i-1}$, then the attacker may easily create a collision. But such collisions are useless for the attacker, who is trying to distinguish the output from faithful generate requests from random values. So in our context, we consider only faithful generate requests.

Lemma 5. *Let BAD_{VV} be as defined in lemma 3. Let BAD^{FGEN}_{UW} denote the event that (j,d,i) exist, where (d,j) is a faithful generate query and $U_{d,i} = W_j$. Then*

$$\Pr[BAD^{FGEN}_{UW} | \overline{BAD_{VV}}] \leq \frac{Q_3}{2^{|V|} - Q}.$$

Proof. Assume (d,i) to be a faithful generate request, and $U_{d,i} \notin \{W_1, \ldots, W_{j-1}\}$. Consider the probability of the attacker to choose W_j with $W_j = U_{d,i}$. Essentially, the attacker has to guess $V_{d,j-1}$. Assume $\overline{\text{BAD}_{\text{VV}}} \wedge \overline{\text{BAD}^{\text{INIT}}_{\text{UW}}} \wedge \overline{\text{BAD}^{\text{RSD}}_{\text{UW}}}$. Also assume that, up to this point, $\text{BAD}^{\text{FGEN}}_{\text{UW}}$ did not yet happen.
In this case, the attacker is guessing an unknown $|V|$ bit output from a challenger's `XOF` query. Tanking into account $U_{d,i} \notin \{W_1, \ldots, W_{j-1}\}$, the probability to guess $U_{d,i}$ when choosing W_j is $\Pr[W_j = U_{d,i}] \leq \frac{1}{2^{|V|} - Q_3}$.
Since there may be up to λ_1 faithful generate requests (d,i) with the same additional input $\alpha_{d,i} = \text{PARSE}(U_{d,i}, 2)$, the chance for query j to collide with any faithful

generate request is less or equal $\frac{\lambda_1}{2^{|V|}-Q_3}$. Since the attacker makes Q_3 queries W_j with $\text{PARSE}(W_j) = 3$ in total,

$$\Pr[\text{BAD}_{\text{UW}}^{\text{FGEN}}] \leq \frac{Q_3 \cdot \lambda_1}{2^{|V|} - Q_3} \leq \frac{Q_3 \cdot \lambda_1}{2^{|V|} - Q}.$$

□

No Bad Events. The last lemma we need for the main result is describes the adversarial advantage in the absence of bad events.

Lemma 6.

$$\Pr\left[\hat{b} = b \mid (\overline{BAD_{UW}^{INIT}} \wedge \overline{BAD_{VV}} \wedge \overline{BAD_{UW}^{RSD}} \wedge \overline{BAD_{UW}^{FGEN}})\right] = \frac{1}{2}$$

Proof. By the definition of the attack game (cf. alg. 3), all the outputs sent to the attacker which depend on b, are from faithful generate requests.
The event $\overline{\text{BAD}_{\text{VV}}}$ implies that, since all the challenger's states $V_{d,i}$ are different, all the `XOF` inputs $U_{d,i}$ to answer generate requests are different. Thus, all the output bits generated in line 3 of the attack game stem from calls $\texttt{XOF}(U_{d,i}, \ldots)$ with different inputs $U_{d,i}$. Depending on b, a faithful generate request will return either of the following values to the attacker:

- By the definition of the `XOF`, cf. Algorithm 1, all the output bits from a `XOF` query are uniformly distributed random bits. I.e., if $b = 0$, the challenger will compute $T \leftarrow\$ \{0,1\}^{\ell+|V|}$ in the `XOF` query and the attacker will see the rightmost ℓ bits of T.
- If $b = 1$, the attacker is given $Z \leftarrow\$ \{0,1\}^{\ell}$.

Regardless of b, the distribution of ℓ-bit values visible for the attacker is exactly the same: the uniform distribution. The only way for the attacker to distinguish $b = 0$ from $b = 1$ is by making a matching query for $\texttt{XOF}(U_{d,i}, \ldots)$: if $b = 0$ it will get the same bits again, if $b = 1$ it will get independent random bits. The event $\text{BAD}_{\text{UW}}^{\text{FGEN}}$ is actually the event of such a matching query being made. □

Proof of theorem 1. According to lemma 6,

$$\Pr\left[\hat{b} = b \mid (\overline{\text{BAD}_{\text{UW}}^{\text{INIT}}} \wedge \overline{\text{BAD}_{\text{VV}}} \wedge \overline{\text{BAD}_{\text{UW}}^{\text{RSD}}} \wedge \overline{\text{BAD}_{\text{UW}}^{\text{FGEN}}})\right] = \frac{1}{2}.$$

I.e., our advantage ϵ in distinguishing $b = 0$ from $b = 1$ is at most the probability to trigger one of the bad events. Hence ϵ is at most the sum of the the bounds from lemmas 2, 3, 4, and 5:

$$\epsilon \leq \frac{Q_1 \cdot \lambda_1}{2^{H_{\text{init}}} - Q} + \frac{\lambda_2}{2^{H_{\text{init}}}} + \frac{R^2}{2 \cdot 2^{|V|}} + \frac{Q_2}{2^{H_{\text{rsd}}} - Q} + \frac{Q_3 \cdot \lambda_1}{2^{|V|} - Q}.$$

We can simplify this to the claimed bound

$$\epsilon \leq Q\left(\frac{\lambda_1}{2^{H_{\text{init}}} - Q} + \frac{1}{2^{H_{\text{rsd}}} - Q}\right) + \frac{\lambda_2}{2^{H_{\text{init}}}} + \frac{R^2}{2 \cdot 2^{|V|}}$$

by applying $\frac{Q_1 \cdot \lambda_1}{2^{H_{\text{init}}} - Q} + \frac{Q_3 \cdot \lambda_1}{2^{|V|} - Q} \leq \frac{Q \cdot \lambda_1}{2^{H_{\text{init}}} - Q}$ (since $H_{\text{init}} \leq |V|$ and $Q_1 + Q_3 \leq Q$) and $\frac{Q_2}{2^{H_{\text{rsd}}} - Q} \leq \frac{Q}{2^{H_{\text{rsd}}} - Q}$ (since $Q_2 \leq Q$). □

6 Matching Attacks

In this section, we will sketch attacks which closely match our claimed security bounds. For the case of simplicity, we do not put any constraints on the usage of the additional input – the attacker is even free to choose all the $a_{d,i}$ as the empty string. This is the case of corollary 1.
We assume each $\mathcal{D}_{d,i}^h$ to be the uniform distribution of h-bit values. This tightly matches the claimed min-entropy h for the $\mathcal{D}_{d,i}^h$.

6.1 Classical Attacks

Attack 1 ($R \approx 2 * 2^{H_{\text{init}}/2}$): The core idea for this attack is to maximise the probability of the event BAD_{VV} and then to exploit it. Note that $H_{\text{init}} \leq |V|$.
The attacker makes $R/2$ instantiate-requests. After each instantiate-request, it makes one request to generate $\ell \gg H_{\text{init}}/2$ output bits.
If $b = 0$, the attacker can expect two of the ℓ-bit outputs to be identical: The seeds for the instantiate-requests are uniformly distributed H_{init}-bit values, with probability $> 1/2$ two of the R seeds will be identical. Thus, these two instantiate-requests will initiate the same output state, which will then be used by the subsequent generate requests to generate the same ℓ-bit output.
If $b = 1$, then, since $\ell \gg H_{\text{init}}/2$, the probability for any two independent ℓ-bit values to be identical is negligible.

Attack 2 ($Q \approx 2 * \max(2^{H_{\text{init}}}/R, 2^{H_{\text{rsd}}})$): The core idea for this attack is to try to guess one of the challenger's `XOF` inputs $U_{d,i}$. We split the attack into two subcases:

Attack 2a ($2^{H_{\text{init}}}/R > 2^{H_{\text{rsd}}}$, $Q \approx 2 * 2^{H_{\text{init}}}/R$): Similarly to attack 1, the attacker makes $R/2$ instantiate requests, interleaved with $R/2$ requests to generate $\ell \gg H_{\text{rsd}}$ output bits each. In contrast to attack 1, the attacker now chooses $Q/2$ random seeds $S_1, \ldots, S_{Q/2} \in \{0,1\}^{H_{\text{rsd}}}$ and picks the ℓ rightmost bits from each of the Q strings $\texttt{XOF}(\texttt{XOF}(S_j, |V|), |V| + \ell)$. With significant probability, one of the attacker's random states S_j will collide with one of the challenger's states $S_{d,i}$.
If $b = 0$, the collision of the attacker's S_j with one of the challenger's states implies the same ℓ-bit output.
If $b = 1$, the probability for one of the attacker's $Q/2$ ℓ-bit output strings with one of the challengers $R/2$ ℓ-bit output strings is negligible, since $\ell > H_{\text{rsd}}$.

Attack 2b ($2^{H_{\text{init}}}/R < 2^{H_{\text{rsd}}}$, $Q \approx 2{*}2^{H_{\text{rsd}}}$): Consider a sequence of a corruption followed by a reseed request and then a request to generate $\ell \gg H_{\text{rsd}}$ output bits. Thanks to the corruption, the attacker knows the input state $V_{d,i}$ for reseed. The input state $V_{d,i+1}$ for generate is computed by $V_{d,i+1} = \texttt{XOF}((V_{d,i} \parallel S_{d,i} \parallel \alpha_{d,i}), |V|)$ from the unknown seed $S_{d,i} \leftarrow\!\$\, \mathcal{D}_i^{H_{\text{rsd}}}$.
If $b = 0$, the visible output consists of the ℓ rightmost bits from $\texttt{XOF}(V_{d,i+1}, |V| + \ell)$. By trying out all $2^{H_{\text{rsd}}}$ choices for $S_{d,i}$, the attacker can find $V_{d,i+1}$ with matching output bits.
If $b = 1$, the visible output consists of ℓ random bits. Since $\ell \gg H_{\text{rsd}}$, the probability for the existence of any $S \in \{0,1\}^{H_{\text{rsd}}}$, such that the ℓ rightmost bits from $\texttt{XOF}(\texttt{XOF}(V_{d,i} \parallel S, |V|), |V| + \ell)$ match ℓ random bits is negligible.

6.2 Quantum Security: Applying Grover's Algorithm

What happens if the attacker can use a quantum computer? In the current paper, we always assume the DRBG to run on a classical computer. By implication, the challenger is classical, and attack 1 still applies. As the attacker can use a quantum computer, it can make `XOF` calls *in superposition*. This allows the quantum attacker to use Grover's algorithm to speed-up attack 2a from $Q \approx 2^{H_{\text{init}}}/R$ `XOF` calls down to $Q \approx 2^{H_{\text{init}}/2}/\sqrt{R}$ and attack 2b from $Q \approx 2^{H_{\text{rsd}}}$ calls down to $Q \approx 2^{H_{\text{rsd}}/2}$. *This is a serious issue for quantum secure DRBGs.*
On the other hand, Grover's algorithm doesn't parallelize well. An implementation of attack 2a or 2b, running c *classical* cores in parallel, speeds up by a factor of c. The speed-up of Grover's algorithm from running c quantum cores is only $\sqrt{c}$.
A concrete example: If the classical attack takes time 2^{85} on a single classical core, then 2^{15} classical cores running in parallel suffice to reduce the wall-clock time for the attack to the equivalent of $2^{85}/2^{15} = 2^{70}$ sequential `XOF` calls. If Grover's algorithm takes the same 2^{85} units of time on a single quantum core, we'd need 2^{30} quantum

cores to mount the attack in time $2^{85}/\sqrt{2^{30}} = 2^{70}$. Given the same number of 2^{30} cores, but classical ones, and 2^{70} units of wall-clock time, we could classically exhaust a 100-bit search space. So let us assume the attacker not to be willing to wait for more than the wall-clock time 2^{70} cryptographic operations would take,[8] either on a classical or on a quantum computer. In this case, 85-bit quantum security is as good as 100-bit classical security. In general, a classical security level of $70 + 2t$ bit is equivalent a quantum security level of only $70 + t$ bit.[9]

7 Proposals

7.1 Numerical Examples

Table 1 provides some numerical examples for approximate security levels, depending on the number R of requests and on the entropy for instantiate and reseed. The classical security level is derived from the lower bounds our main result provides, the quantum security level stems from a straigthforward application of Grover's algorithm. E.g., with LR $= 64$, $H_{\text{init}} = 192$, and $H_{\text{rsd}} = 128$, the classical security level is 128, and the quantum security level is just 64. We always assume $|V| \geq H_{\text{init}}$.

Table 1: Approximate security levels for different instantiations, assuming an ideal `XOF`. The value LR is LR $= \log_2(R)$ if there are no constraints on the choice of the additional input (cf. corollary 1) and LR $= R_1$ if each device is *personalized* by employing a unique name (cf. corollary 3). The connection of Q and $\mathcal{L}$ is explained in section 2.3.
The classical security levels are based on the lower bounds from our main theorem, the quantum security bounds on the application of Grover's algorithm.

H_{init}	LR	H_{rsd}	approximate security level $\mathcal{L}$	
			classical	quantum
192	64	128	128	64
220	50	170	170	85
240	48	192	192	96
240	48	240	192	96
320	64	256	256	128

7.2 Recommendation

We recommend to personalize all implementations of the `XDRBG` *at least* when these require a bound $LR = 50$ or smaller to maintain their security claims. Alternatively, if the `XOF` itself provides a personalization option, as, e.g., cSHAKE does for the SHAKE `XOF`, one can make use of that option and leave the α empty.
In Appendix A.2 we briefly discuss the approach of going beyond personalization or randomization by actually providing additional entropy for the additional input.

7.3 Proposals based on `SHAKE128` and `SHAKE256`

We propose three `XDRBG` instances in Table 2. The first one employs `SHAKE128` with a capacity of 256 bit and provides 128 bits of classical security and 64 bits of quantum security. This matches category one (the lowest category) from the NIST post-quantum security criteria, see Appendix D. The second and third instance employ `SHAKE256`. By restricting the number of requests to $R \leq 2^{58}$, we can claim security category three for the second instance. For $R \leq 2^{64}$, as for the other two instances,

[8] We argue that this assumption is realistic. No attacker cares about 1 000 or 10 000 years to mount an attack on sequential hardware. The attacker cares about the amount of parallel hardware needed to finish the attack in a given amount of time. The threshold size 2^{70} is, of course, open for debate.

[9] Reality may be even worse for the quantum attacker. Evaluating a cryptographic primitive on quantum circuits should be slower, in practice, than evaluating the same primitive on classical hardware.

we could only claim security category two for the second instance. The third instance matches security category five, the top category.

Table 2: Three proposals for DRBG standards and their approximate security levels. The first assumes `SHAKE128` with its 256-bit capacity, the second and third `SHAKE256` with its 512-bit capacity. The value LR is $\mathrm{LR} = \log_2(R)$ if we impose no restrictions on the usage of the additional input, and $\mathrm{LR} = \log_2(R_1)$ if we personalize each device by employing a unique name. We set $|V| =$ capacity. The *category* refers to the NIST post-quantum criteria, cf. Appendix D.

	capacity	H_{init}	LR	H_{rsd}	promised security level $\mathcal{L}$		
					classical	quantum (Grover)	category
`XDRBG-128`	256	192	64	128	128	64	1
`XDRBG-192`	512	240	58	240	192	96	3
`XDRBG-256`	512	320	64	256	256	128	5

To limit the damage from a state compromise, we recommend a limit `maxout` ≈ 2048 bit on the maximum output length from one single generate call.[10] That is, each GENERATE call must return no more than `maxout`bit of output. To generate X bit of output, one has to call the generate function $\lceil X/\texttt{maxout} \rceil$ times. While this has no practical effect on our security bounds, a not-too-large `maxout` limit is a low-cost defense against the impact of a state compromise.

These proposals are inspired by existing or drafted standards. In SP 800-90, a DRBG has a security level, $k \in \{128, 192, 256\}$.[11] The min-entropy needed for instantiation is defined in terms of the security level: $H_{\mathrm{init}} \geq 3k/2, H_{\mathrm{rsd}} \geq k$. I.e., `XDRBG-128` and `XDRBG-256` match the cases $k = 128$ and $k = 256$, respectively. `XDRBG-192` is inspired by the revised version of AIS 20/31 [36], (still a draft as of this writing), which defines lower-limits on effective internal state size and entropy provided to the DRNG[12]. The current draft of AIS 20/31 requires 240 bits of min-entropy for instantiation, so $H_{\mathrm{init}} \geq 240, H_{\mathrm{rsd}} \geq 240$. An implementation meant to comply with both would simply choose the maximum of the two required values. It is always allowable to incorporate *more* entropy than required, or to assume/require a *smaller* number R of requests. In this way, an implementation can be compatible with both standards.

7.4 Alternative Proposals Based on the ASCON Permutation

Recently, [34], NIST has announced plans to standardize ASCON [15], a lightweight family of authenticated encryption and hashing algorithms based on a 320-bit permutation. We anticipate corresponding standards for ASCON-based `XOF`s.

A typical constraint for *lightweight* primitives is the number of input and output bits – e.g., $b = 320$ bit for the ASCON permutation, in contrast to 1600 bit for the `SHAKE` permutation. As pointed out above, any sponge-based `XOF` with a capacity of c bit can only provide $c/2$ bits of classical and $c/3$ bits of quantum security. Note that the rate $r = b - c$ determines the maximum number of input bits to be absorbed or output bits to be squeezed from each time the permutation is called. The performance of a `XOF` is roughly proportional to the rate.

A natural choice for a `XOF` employing the ASCON permutation is the capacity $c = 256$ and thus the rate $r = 64$. But this only allows us to claim at most 85 bits of quantum security. If we employ the ASCON permutation with a capacity $c = 288$, the rate (and thus the performance) goes down to $r = 32$, but now we can claim 144 bits of classical and 96 bits of quantum security.

Table 3 proposes three lightweight variants of the `XDRBG`. The first one, `XDRBG-L-128`, is a lightweight alternative to `XDRBG-128` with the same security claims. The

[10] For efficiency reasons, a value `maxout` may be preferable, which is a multiple of the internal block size or rate of the `XOF` at hand. E.g., the rate of SHAKE-256 is 1088, and we recommend `maxout` $= 2176 = 2 * 1088$.

[11] We expect a currently supported 112-bit security level to be removed from the next version.

[12] Recall that DRNG is the term used in AIS 20/31 for what we refer to as a DRBG.

Table 3: Three proposals for lightweight DRBG standards and their approximate security. The value LR is $\mathrm{LR} = \log_2(R)$ if we impose no restrictions on the usage of the additional input, and $\mathrm{LR} = \log_2(R_1)$ if we personalize each device by employing a unique name.
We set $|V| =$ capacity. The *category* column refers to the NIST post-quantum criteria, cf. Appendix D. Note that the security level of XDRBG-L-144-96 only suffices for classical category two, but its resistance to quantum attacks is at category three.

	capacity	H_{init}	LR	H_{rsd}	promised security level $\mathcal{L}$		
					classical	quantum (Grover)	category
XDRBG-L-128	256	192	64	128	128	64	1
XDRBG-L-128-85	256	220	50	170	128	85	2
XDRBG-L-144-96	288	256	64	192	144	96	2(Q=3)

remaining two are the attempt to provide improved security, especially quantum security, using the same 320-bit permutation. Note that XDRBG-L-128-85 is only able to maintain the desired security level by restricting $\mathrm{LR} \leq 50$, instead of $\mathrm{LR} \leq 64$. We thus recommend every device running XDRBG-L-128-85 to personalize the XDRBG by unsing a unique name for each device. Both XDRBG-128 and XDRBG-128-85 employ a lightweight sponge-based XOF with 256-bit capacity. Using the same permutation, but employing a different sponge-based XOF with a capacity of 288 bit, XDRBG-L-144-96 provides a classical security of 144 bit and a quantum security of 96 bit.

7.5 Performance and Usefulness

XDRBG is designed to use the normal XOF interface, rather than having access to any internal values or state, or making any assumptions about the underlying XOF's inner workings other than its security strength. This makes the DRBG somewhat less efficient, but with the benefit that the DRBG can be implemented pretty efficiently with normal access to a XOF primitive such as SHAKE256, and will work as well for future XOFs–even ones not based on a sponge construction.
XDRBG is designed to keep the inputs to the XOF as small as possible, given its security requirements. Specifically, if r is the *rate* of a sponge-based XOF, the XDRBG can absorb or emit up to $\rho = r - |V|$ bits per permutation call.
Consider the number of permutation calls to handle the different types of requests:

- Handling a reseed request with a seed of length σ and additional input of length $|\alpha|$ needs to make $\lceil (|V| + \sigma + |\alpha| + |\textsc{encode}|)/r \rceil$ permutation calls to absorb its entire input and to emit the first r bits of the new state. If $|V| > r$, it needs an additional $\lceil |V|/r \rceil - 1$ permutation calls to generate the new state in full. In total, this makes

$$\left\lceil \frac{|V| + \sigma + |\alpha| + |\textsc{encode}|}{r} \right\rceil + \left\lceil \frac{|V|}{r} \right\rceil - 1 \quad \text{permutation calls.}$$

- Handling an instantiate request is almost the same as handling a reseed request, except that the input does not expect a seed of any size. Thus, it requires

$$\left\lceil \frac{\sigma + |\alpha| + |\textsc{encode}|}{r} \right\rceil + \left\lceil \frac{|V|}{r} \right\rceil - 1 \quad \text{permutation calls.}$$

- Similarly, handling a generate request for ℓ bits of output takes

$$\left\lceil \frac{|V| + |\alpha| + |\textsc{encode}|}{r} \right\rceil + \left\lceil \frac{|V| + \ell}{r} \right\rceil - 1 \quad \text{permutation calls.}$$

As a concrete (arguably quite typical) example, consider $\sigma = 512$, $|\textsc{encode}| = 8$ (cf. appendix B.1) and $\ell = 256$. We assume the additional input not to be too long, say, $|\alpha| \leq 180$. The performance of our proposed XDRBG variants is as follows:

- XDRBG-128 has rate $r = 1088$ and $|V| = 256$. It only calls the permutation once for each request.
- XDRBG-192 and XDRBG-256 have rate $r = 576$ and $|V| = 512$. Depending on $|\alpha|$ and the request at hand, both XDRBG-192 and XDRBG-256 make between one and three permutation calls for each request.
- The rate and state size of XDRBG-L-128 and XDRBG-L-128-85 is $r = 64$, and $|V| = 256$. In this case, each request requires 12 to 18 permutation calls.
- For XDRBG-L-144-96 the rate is $r = 32$ and the state size is $|V| = 288$. This implies 25 to 39 permutation calls for each request.

8 Design Alternatives

We considered and rejected a number of design alternatives for XDRBG.

8.1 Sponge-based vs XOF-based

The first design choice we had was whether to base the DRBG on a sponge function or a XOF. (Recall that a XOF provides a particular kind of functionality that can be implemented by sponge function, but might also be implemented in some other way.) As discussed above, several prior works, have proposed cryptographic PRNGs based on a sponge construction (specifically using a large ideal permutation), and it would have been relatively easy to adapt those to the requirements of SP 800-90A. However, we believe that basing a DRBG on a XOF provides more flexibility. XDRBG can be based on any XOF, regardless of its underlying structure or assumptions. A future XOF whose security does not rely on an ideal permutation assumption or even use a sponge construction will still work with XDRBG.

8.2 Reseed Interval

SP 800-90A defines a *reseed interval* for its DRBGs. This requires that the DRBG be reseeded after a certain number of GENERATE calls. A relatively small reseed interval provides a defense against the impact of a DRBG state compromise. However, a reseed interval short enough to provide substantial protection from state compromise would also make the DRBG algorithm unworkable in many environments. For example, SP 800-90C defines the RBG1 construction, which has access to live entropy only for instantiation; a similar construction is permitted as a DRG.3 under AIS 20/31. If XDRBG required reseeding every ten or even one hundred GENERATE calls, it would be unworkable for use in these constructions. On the other hand, the huge reseed intervals (requiring a RESEED every 2^{32} or 2^{48} GENERATE calls) in [3] are not too costly, but their security benefit is negligible. For these reasons, we elected not to define a reseed interval as part of XDRBG.

However, we recommend that in any application where it is practical, XDRBG (or any other DRBG) should be reseeded periodically. As described in [25, 9], a reseed *must* wait until sufficient entropy is available to avoid the iterative guessing attack / premature next condition.

A conditional reseed may serve as an easy alternative to a fixed reseed interval without forcing the application to wait (**if** H_{rsd} bits of min-entropy are available, **then** reseed first, continue second **else** continue without reseeding). Alas, this also introduces a potential side-channel vulnerability: The attacker might observe if, whenever the DRBG executes a conditional reseed, the DRBG actually reseeds or not. Thus, each time a conditional reseed is called, one bit of Shannon entropy may be lost.

8.3 Stronger Attack Model

A variation of the attack game could allow the attacker to compromise the DRBG state by *overwriting* it, or *resetting* it to a fixed inital state, rather than by just

reading it. The attack game would otherwise be the same, including the role of the flag `corrupt`. But now, the attacker could benefit from a multitarget attack, similar to the attack from section 2.5: Fix a state V^* and repeat the following three-step sequence as often as possible: (1) set the DRBG to V^*, (2) reseed, and (3) generate some output bits. Eventually try $O(Q)$ times to guess any of the $O(R)$ seeds from step (2) and generate the output bits to detect a match.

As pointed out above, we doubt the plausibility of an attacker to the DRBG state by a chosen or known value V^*. But for readers who prefer to consider such attacks, we point out the following: Reseeding is never worse than instantiating, except that H_{rsd} may be smaller than H_{init}. In fact, the combination of setting the DRBG state to V^* and then to reseed with a seed S is equivalent to instantiating the DRBG state with a seed $S^* = V^* \parallel S$. Since V^* is known to (or even chosen by) the attacker, the distributions, which S and S^* are drawn from, have exactly the same min-entropy. Accordingly, we make two recommendations for the stronger attack model: (1) set $H_{\text{rsd}} \approx H_{\text{init}}$, and (2) if the `XDRBG` is personalized, then personalize both instantiate and reseed requests. In that context, observe that AIS 20/31 sets $H_{\text{init}} = H_{\text{rsd}}$.

Regarding the first recommendation, we revisited the proof for our main result. As it turned out, both proof and result still apply, with a slightly tweaked bound. In fact, for $H = H_{\text{init}} = H_{\text{rsd}}$ we can replace equation 2 by

$$\epsilon \leq Q\left(\frac{\lambda_1}{2^H - Q} + \frac{1}{2^H - Q}\right) + \frac{\lambda_2}{2^H} + \frac{R^2}{2 \cdot 2^{|V|}},$$

which also applies to the attack model where the attacker can set the DRBG state.

9 Conclusions

Drawing on previous work in [8, 11], we have proposed a new class of DRBG that can be based on any `XOF`, and analyzed its security in a model well adapted for DRBGs as defined in [3]. We have kept the specification of `XDRBG` general enough to adapt to the different requirements of SP 800-90 and AIS 20/31, and to work for any `XOF`. It is possible to make a more efficient DRBG by altering the internal workings of a sponge-based `XOF`, but this would result in a less generally-useful DRBG. We prefer to provide a more generic design.

`XDRBG` is quite efficient. Assuming a reasonable-length seed and output length, every `XOF` query made by `XDRBG` with `SHAKE128` or `SHAKE256` will result in only a single permutation call. And even `XDRBG`-L144-96, the lightweight variant based on a permutation over only 320 bits, where we squeezed in almost as much quantum security as anyhow possible (96 bits), can process typical requests by calling the permutation less than fifty times for each request.

Our hope is that `XDRBG` will be a useful addition to the set of DRBGs currently in use. The goal of our concrete perimeter sets is to support security engineers to make well-informed decisions about state size and entropy requirements.

Open Questions. We envision least two interesting directions for future research:

1. Our analysis for quantum security focused on the application of Grover's algorithm. Also, recall the cautionary note from Footnote 4 regarding the quantum security of sponge-based `XOF`s. Nevertheless, we conjecture that these bounds actually describe the security of the `XDRBG` against quantum adversaries very well. A proof for those bounds, similar to our classical security analysis, would probably assume the compressed oracle model [40]. We leave such a proof for future work, as it would likely require an entire paper of its own.
2. Our security proof assumes oracle-independent entropy sources. While this seems entirely reasonable for the context of random number generation with trusted entropy sources (as in SP 800–90 or AIS 20/31), it would be interesting to know the security bounds for oracle-dependent sources, using the techniques of [11]. Again, we leave this analysis for future work.

References

[1] Advanced Micro Devices. AMD RNG ESV public use document, document version 0.4. Tech. rep. url = https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/entropy/E27_PublicUse.pdf.

[2] Agievich, S., Marchuk, V., Maslau, A., and Semenov, V. Bash-f: another lrx sponge function. Cryptology ePrint Archive, Paper 2016/587, 2016. https://eprint.iacr.org/2016/587.

[3] Barker, E., and Kelsey, J. Recommendation for random number generation using deterministic random bit generators. Tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2015.

[4] Barker, E., Kelsey, J., McKay, K., Roginsky, A., and Turan, M. S. Recommendation for random bit generator (rbg) constructions (3rd draft). Tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2022.

[5] Bernstein, D. J., Kölbl, S., Lucks, S., Massolino, P. M. C., Mendel, F., Nawaz, K., Schneider, T., Schwabe, P., Standaert, F.-X., Todo, Y., and Viguier, B. Gimli. Submission to the NIST Lightweight Cryptography Standardization Process, 2019. https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates.

[6] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. Sponge functions. *Ecrypt Hash Workshop 2007* (2007).

[7] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. On the indifferentiability of the sponge construction. In *Advances in Cryptology – EUROCRYPT 2008* (Berlin, Heidelberg, 2008), N. Smart, Ed., Springer Berlin Heidelberg.

[8] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. Sponge-based pseudo-random number generators. In *CHES 2010* (Santa Barbara, CA, USA, Aug. 17–20, 2010), S. Mangard and F.-X. Standaert, Eds., vol. 6225 of *LNCS*, Springer, Heidelberg, Germany, pp. 33–47.

[9] Coretti, S., Dodis, Y., Karthikeyan, H., Stephens-Davidowitz, N., and Tessaro, S. On seedless prngs and premature next. In *3rd Conference on Information-Theoretic Cryptography, ITC 2022, July 5-7, 2022, Cambridge, MA, USA* (2022), D. Dachman-Soled, Ed., vol. 230 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 9:1–9:20.

[10] Coretti, S., Dodis, Y., Karthikeyan, H., Stephens-Davidowitz, N., and Tessaro, S. On seedless prngs and premature next. *IACR Cryptol. ePrint Arch.* (2022), 558.

[11] Coretti, S., Dodis, Y., Karthikeyan, H., and Tessaro, S. Seedless Fruit is the sweetest: Random number generation, revisited. In *CRYPTO 2019, Part I* (Santa Barbara, CA, USA, Aug. 18–22, 2019), A. Boldyreva and D. Micciancio, Eds., vol. 11692 of *LNCS*, Springer, Heidelberg, Germany, pp. 205–234.

[12] Coretti, S., Dodis, Y., Karthikeyan, H., and Tessaro, S. Seedless fruit is the sweetest: Random number generation, revisited. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I* (2019), A. Boldyreva and D. Micciancio, Eds., vol. 11692 of *Lecture Notes in Computer Science*, Springer, pp. 205–234.

[13] Czajkowski, J. Quantum indifferentiability of SHA-3. *IACR Cryptol. ePrint Arch.* (2021), 192.

[14] Dobraunig, C., Eichlseder, M., Mangard, S., Mendel, F., Mennink, B., Primas, R., and Unterluggauer, T. ISAP v2.0. *IACR Trans. Symm. Cryptol. 2020*, S1 (2020), 390–416.

[15] Dobraunig, C., Eichlseder, M., Mendel, F., and Schläffer, M. Status Update on Ascon v1.2. Update to the NIST Lightweight Cryptography Standardization Process, 2022. https://csrc.nist.gov/csrc/media/Projects/lightweight-cryptography/documents/finalist-round/status-updates/ascon-update.pdf.

[16] Dodis, Y., Gennaro, R., Håstad, J., Krawczyk, H., and Rabin, T. Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In *CRYPTO 2004* (Santa Barbara, CA, USA, Aug. 15–19, 2004), M. Franklin, Ed., vol. 3152 of *LNCS*, Springer, Heidelberg, Germany, pp. 494–510.

[17] Dodis, Y., Vaikuntanathan, V., and Wichs, D. Extracting randomness from extractor-dependent sources. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I* (2020), A. Canteaut and Y. Ishai, Eds., vol. 12105 of *Lecture Notes in Computer Science*, Springer, pp. 313–342.

[18] Ferguson, N. The Windows 10 random number generation infrastructure. Tech. rep., Microsoft, 2019. url = https://aka.ms/win10rng.

[19] Ferguson, N., and Schneier, B. *Practical cryptography*. Wiley, 2003.

[20] Gaži, P., and Tessaro, S. Provably robust sponge-based PRNGs and KDFs. Cryptology ePrint Archive, Report 2016/169, 2016. https://eprint.iacr.org/2016/169.

[21] Gazi, P., and Tessaro, S. Provably robust sponge-based PRNGs and KDFs. In *EUROCRYPT 2016, Part I* (Vienna, Austria, May 8–12, 2016), M. Fischlin and J.-S. Coron, Eds., vol. 9665 of *LNCS*, Springer, Heidelberg, Germany, pp. 87–116.

[22] Hutchinson, D. A robust and sponge-like prng with improved efficiency. Cryptology ePrint Archive, Paper 2016/886, 2016. https://eprint.iacr.org/2016/886.

[23] Hutchinson, D. A robust and sponge-like PRNG with improved efficiency. In *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers* (2016), R. Avanzi and H. M. Heys, Eds., vol. 10532 of *Lecture Notes in Computer Science*, Springer, pp. 381–398.

[24] Kelsey, J., Schneier, B., and Ferguson, N. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography, 6th Annual International Workshop, SAC'99, Kingston, Ontario, Canada, August 9-10, 1999, Proceedings* (1999), H. M. Heys and C. M. Adams, Eds., vol. 1758 of *Lecture Notes in Computer Science*, Springer, pp. 13–33.

[25] Kelsey, J., Schneier, B., Wagner, D., and Hall, C. Cryptanalytic attacks on pseudorandom number generators. In *FSE'98* (Paris, France, Mar. 23–25, 1998), S. Vaudenay, Ed., vol. 1372 of *LNCS*, Springer, Heidelberg, Germany, pp. 168–188.

[26] Killmann, Wolfgang and Schindler, Werner. A proposal for functionality classes for random number generators. Tech. Rep. AIS20, Bundesamt für Sicherheit in der Informationstechnik (BSI), 2011.

[27] Mechalas, J. P. Intel® Digital Random Number Generator (DRNG) Software Implementation Guide, 2018. https://www.intel.com/content/www/us/en/developer/articles/guide/intel-digital-random-number-generator-drng-software-implementation-guide.html.

[28] Müller, S., Mayer, S., Holz auf der Heide, C., and Hohenegger, A. Documentation and analysis of the Linux random number generator. Tech. rep., Bundesamt für Sicherheit in der Informationstechnik (BSI), 2023.

[29] National Institute of Standards and Technology. Advanced encryption standard (AES). Tech. Rep. Federal Information Processing Standards (FIPS) Publication 197, U.S. Department of Commerce, Washington, D.C., 2001.

[30] National Institute of Standards and Technology. Digital signature standard (dss). Tech. Rep. Federal Information Processing Standards (FIPS) Publication 186-4, U.S. Department of Commerce, Washington, D.C., 2013.

[31] National Institute of Standards and Technology. Secure hash standard (SHS). (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 180-4, August 2015. https://doi.org/10.6028/NIST.FIPS.180-4.

[32] National Institute of Standards and Technology. Sha-3 standard: Permutation-based hash and extendable output functions. Tech. Rep. Federal Information Processing Standards (FIPS) Publication 202, U.S. Department of Commerce, Washington, D.C., 2015.

[33] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. url = https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[34] National Institute of Standards and Technology. Lightweight cryptography standardization process: NIST selects Ascon, Feb 2023. url = https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon.

[35] Nir, Y., and Langley, A. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018.

[36] Peter, Matthias and Schindler, Werner. A proposal for functionality classes for random number generators–version 2.35 draft. Tech. Rep. AIS20, Bundesamt für Sicherheit in der Informationstechnik (BSI), 2023.

[37] Rivest, R. L., and Schuldt, J. C. N. Spritz - a spongy rc4-like stream cipher and hash function. *IACR Cryptol. ePrint Arch.* (2016), 856.

[38] Turan, M. S., Barker, E., Kelsey, J., McKay, K., Baish, M., and Boyle, M. Recommendation for random number generation using deterministic random bit generators. Tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2018.

[39] Woodage, J., and Shumow, D. An analysis of NIST SP 800-90A. In *EUROCRYPT 2019, Part II* (Darmstadt, Germany, May 19–23, 2019), Y. Ishai and V. Rijmen, Eds., vol. 11477 of *LNCS*, Springer, Heidelberg, Germany, pp. 151–180.

[40] Zhandry, M. How to record quantum queries, and applications to quantum indifferentiability. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II* (2019), A. Boldyreva and D. Micciancio, Eds., vol. 11693 of *Lecture Notes in Computer Science*, Springer, pp. 239–268.

A Unusual Use Cases

Our security analysis above discusses the normal use cases for a DRBG. In this appendix, we consider less common ways a DRBG may be used and how this might affect our security bounds.

A.1 Seeding from another DRBG

In some contexts, a DRBG's seed may come from another DRBG. Since our security proof assumes access to entropy for seed material, it is natural to consider the security impact of seeding from a DRBG. We can extend our security bounds to deal with the

situation, by simply incorporating an additional term for violating the security of the DRBG providing the seed. Informally, if the attacker cannot distinguish the outputs of the DRBG providing the seed from ideal random outputs, it also cannot gain any advantage in distinguishing XDRBG outputs seeded from those DRBG outputs.

A.2 Adding Entropy via the Additional Input

XDRBG, like the DRBGs in SP 800–90A, allows for an optional additional input to each DRBG call. This input may be used in many different ways in practical applications. For example:

1. Some systems maintain a *seed file*, to save entropy across device restarts as a hedge against an entropy source failure. A natural way to incorporate the seed file into the DRBG state is to put it into the additional input of the instantiate call.
2. Secret information, such as the hash of a private key, can be incorporated into the DRBG during instantiation, again to provide a hedge against the failure of the entropy source.
3. Additional entropy can be drawn from some secondary entropy source, or even from the primary entropy source, and provided to the DRBG during instantiation or reseeding.

Intuitively, it is easy to see that entropy provided in the additional input is incorporated into the DRBG state in the same way as the seed, since the additional input is simply appended to the seed in instantiate and reseed calls. Thus, an INSTANTIATE or RESEED in which sufficient entropy is provided in the additional input will end up in a secure state, even if no entropy is provided in the seed.
Let h_1 the entropy in the seed, and h_2 be the entropy in the additional input. By virtue of our encoding function, and as long as the seed is independent from the additional input, the string input to the XOF must thus have h_1+h_2 bits of min-entropy. And even if seed and additional input are statistically dependent, the min-entropy of the encoded input for the XOF has at least $\max(h_1, h_2)$ bit of min entropy.

B The Function encode

Note that we need a function $\text{ENCODE} : \{0,1\}^* \times \{0,1\}^* \times \{0,1,2\} \to \{0,1\}^*$ such that for all $(S, \alpha, n) \neq (S', \alpha', n')$ $\text{ENCODE}(S, \alpha, n) \neq (S', \alpha', n')$. Note that the length of $\text{ENCODE}(S, \alpha, n)$ must exceed the length of $S \parallel \alpha$. We refer to the stretch as $|\text{ENCODE}| = |\text{ENCODE}(S, \alpha, n)| - |S| - |\alpha|$. The stretch may be constant, or depend on S, α, and n.
There are many ways to define a suitable encoding functions. We also may impose constraints on the seed length $|S|$ or on the length $|\alpha|$ of the additional input and define ENCODE under such a constraint.

B.1 An Encoding with a Single-Byte Stretch

Restrict the length $|\alpha|$ of α (in bit) to $|\alpha| \in \{8*0, 8*1, \ldots, 8*84\}$. I.e., α is between zero and 84 bytes long. Define

$$\text{ENCODE}(S, \alpha, n) = (S \parallel \alpha \parallel (n * 85 + |\alpha|/8)_8),$$

where $(\ldots)_8$ indicates an 8-bit (i.e., single-byte) encoding of a value in $\{0, 254\}$. Thus, $|\text{ENCODE}| = 8$, i.e., the stretch is constantly one byte.

B.2 A More Flexible Encoding, Optimized for Personalization

Sometimes, the constraint for α to be at most 84 bytes long may be onerous. We thus propose to enhance the encoding presented above, to handle additional inputs of lengths $|\alpha| \notin \{0, 8*1, \ldots, 8*84\}$: define a hash function

$$h : \{0,1\}^* \times \{0,1\}^{h_{\text{hash}}*8}, \quad h(x) = \texttt{XOF}(x \parallel 255_8, h_{\text{hash}} * 8)$$

generating h_{hash}-byte hashes, and to feed $h(\alpha)$ into the encoding instead of α directly. Note that the hash function does not need to be collision resistant (cf. the remark at page 11), though we consider resistance against 2nd preimages and k-collisions with $k \gg 2$ as desirable. For concreteness, we propose $h_{\text{hash}} * 8 \geq |V|/1.6$ and define the encoding function as follows. If $|ai| \in \{0, 8, \ldots, 8*83\}$ we can encode α directly:

$$\text{ENCODE}(S, \alpha, n) = (S \parallel \alpha \parallel (n * 85 + |\alpha|/8)_8).$$

Regardless of $|\alpha|$, and especially for $|\alpha| \notin \{0, 8, 8*83\}$:

$$\text{ENCODE}(S, \alpha, n) = (S \parallel h(\alpha) \parallel (n * 85 + 84)_8).$$

Though this may require two `XOF` queries to respond to one DRBG request, our security analysis still applies. The point is that the final byte of every input to the `XOF` serves as a domain separator: for $n \in \{0, 1, 2\}$,

- the byte is $(n*85+\frac{|\alpha|}{8})_8$ if we handle α directly (restricted to $\frac{|\alpha|}{8} \in \{0, 1, \ldots, 83\}$),
- the byte is $(n * 85 + 84)_8$ if we first hash α (maximum value $2 + 85 + 84 = 254$),
- and, finally, the byte is 255_8 for the hash function itself.

This approach has a nice benefit: Lengthy names for personalization come at essentially no cost. Recall that during personalization, we recommend a unique device name α to be chosen. This name is chosen once and used again and again as the additional input each time the device handles a DRBG call. For a typical hash function h, the time to compute $h(\alpha)$ is proportional to $|\alpha|$. Now, the proposed encoding allows to compute $h(\alpha)$ only once, namely when the device is personalized, and then to store $h(\alpha)$ in the device to be used whenever the device handles a DRBG call.

C HashXOF

Although there are already multiple widely-used DRBGs based on a hash function, we can construct a `XOF` suitable for `XDRBG` from any standard hash function, such as SHA256. The design is as follows:

function `HashXOF`(x, ℓ)
  $t \leftarrow$ `Hash`$(x \parallel 0_{64})$
  $Z \leftarrow \varepsilon$
  $i \leftarrow 1$
  while $|Z| < \ell$ **do**
    $Y \leftarrow$ `Hash`$(t \parallel i_{64})$
    $i \leftarrow i + 1$
    $Z \leftarrow Z||Y$
  return(Z truncated to ℓ bits)

Using `XDRBG` with `HashXOF`(SHA256) will provide comparable performance to either HMAC_DRBG(SHA256) or Hash_DRBG(SHA256).

D NIST Post-Quantum Security Categories

NIST defines five security categories for submissions to the Post-Quantum Cryptography process in [33]. These categories are as follows:

category	requirement	security	
	any attack must require computational resources comparable to or greater than those required for	classical	quantum
1	key search on a block cipher with a 128-bit key	128	64
2	collision search on a 256-bit hash function	128	85
3	key search on a block cipher with a 192-bit key	192	96
4	collision search on a 384-bit hash function	192	128
5	key search on a block cipher with a 256-bit key	256	128

E Pool-Based DRNGs

In this paper, we assume the existence of trusted entropy sources that provide strings with a known amount of min-entropy on demand. In this, we follow the lead of NIST and BSI standards–SP 800-90 and AIS-20/31 specify techniques for evaluating entropy sources, and then assume the availability of entropy sources whose claims of entropy can be relied upon.

When entropy sources do not reliably provide a known amount of min-entropy, or when they may even be adversarially controlled, a very different approach is required. The Fortuna cryptographic PRNG, first described in [19], is designed to guarantee that its PRNG algorithm will eventually be seeded securely, as long as the entropy sources used are providing *some* entropy, even without any way to know how much is being provided. This kind of design is modeled in depth in [9].

The Fortuna PRNG considers different entropy pools, which over time receive inputs from one or more sources of entropy. It is not exactly known how much entropy the sources provide. From time to time, the entropy gathered in some subset of these pools is used to update the state. If, after a state compromise, the pools have gathered a sufficient amount of entropy, then the PRNG recovers from the compromise. Otherwise, the entropy from those pools is essentially lost, since the attacker can use his knowledge of the previous state of the RNG and subsequent outputs to guess the entropy input. (This class of attack is referred to as “premature next" in [10] and as the "iterative guessing attack" in [25].) The goal of Fortuna is to guarantee that the PRNG will eventually reach a secure state if there is any entropy being provided to it.

Potentially adversarial entropy sources, as well as “oracle-dependent" entropy sources (entropy sources whose distributions are not independent of the cryptographic functions used to extract entropy from them) are analyzed in [12, 17]. Since we assume trusted, already-analyzed entropy sources in this paper (in keeping with the SP 800-90 and AIS 20/31 standards), we do not consider adversarial or oracle-dependent sources here.

F Addressing Reviewers’s Comments (ToSC 2023 #3)

Reviewers’ Post-Rebuttal Comment:

> *All three reviewers found the paper interesting. As acknowledged by you, some changes are needed before publication: 1) the proof needs to be corrected 2) a more extensive comparison with the state of the art should be included 3) the attack game should be reworked to include the multi-user case.*

Thank you for your interest in our paper! As elaborated below, we did address your requests for changes.

1) Correcting the Proof.

> *Overall, the paper is well-written, but seems to have a small technical flaw (see my comment below). [. . .]*

Indeed, in the analysis of bad event 1, if W_j happens after U_i, the authors should not use the randomness from U_i to bound the collision probability.

In the previous version of the proof, we did indeed not properly distinguish the case of the `XOF` query being made before the matching `XDRBG` request from the case of the query made after the request. We have fixed this issue now.
Note that by this fix, our bounds got slightly worse: We had to replace factors of the form $Qx/2^H$ by $Qx/(2^H - Q)$, where $H \in \{H_{\text{init}}, H_{\text{rsd}}\}$, $x \geq 1$, and Q is the number of `XOF` queries made by the adversary. Fortunately, this change does not significantly affect the security of the `XDRBG`, for the cases of interest, namely for $Q \ll 2^H$. (As a concrete example, consider $H = H_{\text{init}} = 128$ and $Q = 2^{100}$.)

2) More Extensive Comparison with the State of the Art.

It could be interesting to compare the work with state of the art, even if only at qualitative level.

In our view, the state of the art in cryptographic random number generation using trusted entropy sources is shaped by the SP800-90 [3, 38] and AIS 20/31 [26] standards. `XDRBG` is specifically designed with the requirements of these standards in mind. In the paper, we frequently refer to the differences of these standards and their drafted revisions [4, 36] to our proposal, and we discuss the relevance of these standards to the choices we actually made for the `XDRBG`. See sections 1, 2, 4, 7.3, 8, and 9.
The state of the art for random number generation without trusted entropy sources is probably the Fortuna PRNG and related designs, as analyzed by Coretti et al. For a comparison of our approach to pool-based DRNGs, we have added appendix E.

3) Multi-User Attack Scenarios.

I think it is unfortunate that the multi-user analysis (and the impact of AI in that case) is just briefly discussed in Section 6. I believe this should have been directly part of the security game and analysis in section 5 (for example by introducing an additional distribution for AI, and then discussing possible choices). This would make the main result of the paper much more general and powerful.

We have added the multi-user (or, as we put it, multi-device) analysis to the paper. The changes include a revised security game (cf. algorithms 3 and 4) and a new main theorem, which generalizes our previous main theorem by dealing with the revised security game. To describe typical application scenarios, we did introduce three corollaries, Corollary 1 resembles our previous main theorem.

Some Minor Issues we did Address.

To my understanding the current version of 900-80A there is also the security level 112 bit possible.

We have added footnote 11.

To my understanding the current draft of AIS 20/31 (draft 2.35/2.36) requires: min-entropy $\geq$ 240 bits, Shannon entropy $\geq$ 250 bits, so footnote 11 might be removed.

We have fixed this. See also footnote 5.

In Table 1 was missing $H_{init} = H_{rsd} = 240$ bits as the with the AIS20/31 was given in the motivation. Also, I had some problems to link the main result from Eq. (1) to Table 1 as there is no value given for Q.

We have added the missing line in table 1. Furthermore, section 2.3 now elaborates on the relationship of the security level $\mathcal{L}$ and Q.

Supplementary Material:
We will add a latexdiff file to indicate the changes since the previous submission. We will also add a reference implementation (source code `xdrbg.c` and `Makefile`).