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 ramdom source with sufficient entropy. The current paper formalizes a security notion for a DRBG, allowing the attacker to compromise the internal state of the DRBG, requiring the DRBG to maintain the security of generated output bits prior to the compromise, and also requiring the DRBG to recover from a state compromise, once a new random seed becomes available to the DRBG. The paper proposes `XDRBG`, a new DRBG based on any eXtended 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 XOF`. Alternative instantiations suitable for lightweight applications can be based on ASCON.

Keywords: pseudorandom bit generation · forward security · backward security · extended output function (`XOF`)

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 deterministic cryptographic algorithm (called a DRBG in [3] and this work, and called a PRNG, cryptographic PRNG, or DRNG in other works) produces the output bits. See [24, 25, 15, 1, 20] for widely-used examples of this kind of construction.

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: SP800-90A [3] and AIS20/31 [22, 32].

In [6], a new way of constructing hash functions was proposed, called a *sponge function*. A sponge processes data like a cryptographic hash, but can generate arbitrary-length outputs. This broad class of cryptographic primitive (which might also be realized using other constructions) was named a `XOF` *(eXtended Output Function)* in [28]. Informally, a `XOF` works like a cryptographic hash function, but allows for an arbitrary-length output string[2]. The calls $\texttt{XOF}(x, n)$ and $\texttt{XOF}(x, n + u)$ will yield identical results in their first n

[1] Thus, the DRBGs in [3] are based on AES [26], SHA2 [27], or SHA3 [28], and the DRBG currently used in Linux is based on Chacha20 [31].

[2] In principle, a `XOF` could be defined to have a maximum output length, but the `SHAKE` functions allow arbitrary length output.

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 XOFs based on sponge functions such as SHAKE128 and SHAKE256. In recent years, many new sponge-based hash functions have been proposed; typically these also support XOF functionality. For example, see [33, 2, 5, 12, 13].

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.

1.1 Prior Work

In [8], Bertoni et al describe a generic construction for cryptographic random bit generation based on a sponge construction. Additional proposals along these lines were made by Gaži and Tessaro[16, 17], Hutchinson [18, 19]. and Coretti et al [10]. 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, in our analysis, 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, 34, 4] and AIS 20/31 [22, 32], 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 [14] and later [35], rather than the model of [10].) We suspect that the techniques of [10] 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 D be a distribution over strings $\{0,1\}^*$. We write $S \leftarrow\!\$\, D$ if the string S is chosen according to D. In the context of the current paper, we will use the min-entropy

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

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 bit of min-entropy*, we will always get at least h bit 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 shall be drawn from a source with at least H_{init} bit of min entropy.
- H_{rsd}: When the reseed command is called, the seed shall be drawn from a source with at least H_{rsd} bit of min entropy.

2.2 Interface for DRBG

Following [3], we define three DRBG operations:

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

In [3], DRBGs are defined with a particular interface, which assumes the DRBG can draw bits from the entropy source directly, and 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. $\textsc{Instantiate}(S, \text{AI})$ discards the previous state, and the new state only depends on the seed and the optional string AI. $\textsc{Reseed}(V, S, \text{AI})$ refreshes the state based on the previous state, the seed, and the optional string AI. 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. 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 entropy. A DRBG without access to any new seed material (e.g., from an entropy source) simply cannot achieve backward security.

AIS 20/31 has two very similar but slightly weaker required properties for a DRNG, called enhanced forward security and enhanced backward security.

2.5 Multitarget Attack on INSTANTIATE

One goal of this paper is to discuss resistance from a specific kind of time / memory / data multitarget attack.

Suppose some system instantiates its DRBG at each startup with a seed S containing k bits of entropy. After I such instantiations, an attacker can recover one DRBG state with a $2^k/I$ search, assuming that each instantiation produced at least k bis of output. 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 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 Extended 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 a XOF-attacker to make a sequence of queries, and then the XOF-attacker will try to differentiate the XOF from a truly random oracle. Write W for the sum of the lengths of all inputs and outputs (in bit) made by the XOF-attacker. We say, a XOF supports k-bit

[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 $\{0,1\}^* \rightarrow \{0,1\}^*$. 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 bit)

security, if for every `XOF`-attacker, the advantage in distinguishing the `XOF` from random is at most $W/2^k$.

FIPS 202 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 bit and `SHAKE256` 512 bit.

A sponge-based `XOF` with a capacity of c bits can maintain about $c/2$ bit security against classical attackers [7] and about $c/3$ bit against quantum attackers [11].[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`.

3.1 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] Cautionary note: To the best of our knowledge, the claimed $c/3$ bit of quantum security for a sponge with c bit capacity has so far only been published at an eprint server [11], but not yet at a peer-reviewed conference or journal. In the current paper, we assume the claim to be correct.

4. All XDRBG functions support an additional input AI to personalize the DRBG.

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 ($|V|$), maxout, H_{rsd}, or H_{init}. Concrete recommendations for these parameters appear in Section 7.

Algorithm 2 XDRBG Definition

function INSTANTIATE(seed, AI)
  # Returns $|V|$-bit state; source for seed: $\geq H_{init}$ bit min-entropy; AI may be empty.
  $V \leftarrow$ XOF(seed $\|$ AI, $|V|$)
  return(V)
function RESEED(V', seed, AI)
  # Returns $|V|$-bit state; source for seed: $\geq H_{rsd}$ bit min-entropy; AI may be empty.
  $V \leftarrow$ XOF(V' $\|$ seed $\|$ AI, $|V|$)
  return(V)
function GENERATE(V', ℓ, AI)
  # Returns $|V|$-bit state and ℓ-bit string Σ; AI may be empty.
  $T \leftarrow$ XOF(V' $\|$ AI, $\ell + |V|$)
  $V \leftarrow$ first $|V|$ bits of T
  $\Sigma \leftarrow$ last ℓ bits of T
  return(V, Σ)

3.2 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. This led us to a few design decisions:

1. Each call to a DRBG function (INSTANTIATE, RESEED, or GENERATE) results in a single XOF query.

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. 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 its 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 an SP800-90A type DRBG to exploit some weakness in its design. The attacker sends requests to the challenger, ordering it to carry out any sequence of DRBG calls: generate outputs, reseed, instantiate. The only constraint is that the first request must be instantiate. The attacker is free to choose the *additional input* (AI) for the requests. After every request, the attacker can learn the DRBG state by *compromising* it. The DRBG must recover from a compromise when an instantiate or reseed call is made.

The attacker can force the DRBG to repeatedly be instantiated. This captures the multitarget instantiation attack described above, as well as any other weaknesses in the instantiation process of the DRBG might suffer from. By allowing the attacker to repeatedly compromise the state of the DRBG 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. Since the attacker can compromise (i.e., learn) a DRBG state, she can generate certain output bits on her own – until an instantiate or reseed call has been made (see above). 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 [35]. Our game incorporates a wider range of possible attacks, and closely tracks with the assumptions and requirements of the SP800-90 series and AIS20/31:

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 the availability of a properly designed and tested entropy source (known as an NTG, PTG.2 or PTG.3 in AIS20/31) 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.

   SP800-90c requires a security parameter k and fixes the entropy bounds by $H_{\text{init}} = 3k/2$, and $H_{\text{rsd}} = k$. AIS20 requires[5] $H_{\text{init}} = H_{\text{rsd}} = 240$.

[5] Actually, AIS20/31 requires 240 bit of Shannon-entropy in both cases, rather than 240 bit of min-entropy. Since Shannon entropy is never less than min-entropy, these bounds ensure that the DRBG will meet the requirements of AIS20/31.

Algorithm 3 DRBG Security Game:
The attacker wins if the final message it receives from the challenger is a 1.
The number of requests made by the attacker is denoted as R, and the attacker can directly query the `XOF` up to Q times, not counting the challenger's `XOF` queries, made when addressing the R requests.

function CHALLENGER(k)
 # Start the game: randomly choose the secret bit b.
  $b \leftarrow\$ \{0,1\}$
 # Attacker first commits to distributions and then is granted access to `XOF`*.*
  Attacker chooses distributions $D_1^h, \ldots, D_R^h$, as elaborated in the text.
  Challenger grants direct access to `XOF` for attacker, for up to Q queries.
 # The first request is always instantiate.
  Receive request=`R_INST`(AI_1) from attacker.
  $S \leftarrow\$\ D_1^{H_{\text{init}}}$
  $V_1 \leftarrow$ INSTANTIATE(S, AI_1)
  `corrupt` $\leftarrow$ `false`
 # After every request, even after the first one, attacker can compromise.
  if attacker compromises DRBG **then**
    Send DRBG state V_1 to attacker.
    `corrupt` $\leftarrow$ `true`
 # The remaining $R-1$ *requests follow now.*
  for step $\leftarrow \{2, \ldots, R\}$ **do**
    Receive request $\in$ {`R_INST`, `R_OUT`, `R_RESEED`} from attacker.
    if request is `R_OUT`($\ell, \mathrm{AI}_{\text{step}}$) **then**
      $(V_{\text{step}}, Z) \leftarrow$ GENERATE($V_{\text{step}-1}, \ell, \mathrm{AI}_{\text{step}}$)
      if $b = 1$ AND NOT `corrupt` **then**
        $Z \leftarrow\$ \{0,1\}^n$
      Send Z to attacker.
    else if request is `R_RESEED`($\mathrm{AI}_{\text{step}}$) **then**
      $S \leftarrow\$\ D_{\text{step}}^{H_{\text{rsd}}}$
      $V_{\text{step}} \leftarrow$ RESEED($V_{\text{step}-1}, S, \mathrm{AI}_{\text{step}}$)
      `corrupt` $\leftarrow$ `false`
    else if request is `R_INST`($\mathrm{AI}_{\text{step}}$) **then**
      $S \leftarrow\$\ D_{\text{step}}^{H_{\text{init}}}$
      $V_{\text{step}} \leftarrow$ INSTANTIATE($V_{\text{step}-1}, S, \mathrm{AI}_{\text{step}}$)
      `corrupt` $\leftarrow$ `false`
    if attacker compromises DRBG **then**
      Send DRBG state V_{step} to attacker.
      `corrupt` $\leftarrow$ `true`
 # Finish the game: the attacker tries to guess the secret bit b.
  Receive $\hat{b}$ from attacker.
  if $b = \hat{b}$ **then** Send 1 to attacker.
  else Send 0 to attacker.

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

- Also at the beginning of the game, the attacker will specify a sequence of distributions $D_1^h, \ldots, D_R^h$ with min-entropy $h \in \{H_{\text{rsd}}, H_{\text{init}}\}$. As it is formally specified by the attacker, it covers *all* realistic entropy sources.

  Note that a realistic entropy source may change its distribution over time, e.g., due to heating up or cooling down, thus, we allow different D_i^h for different i.
- To restrict the entropy sources to *realistic* ones, we require the attacker to commit to all distributions D_i^h 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 3–5 of algorithm 3).

  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`.

3. During the attack game, the attacker makes *exactly* R requests:

   - The first request is always `R_INST`.
   - For each of the remaining ones the attacker can choose between `R_INST`, `R_OUT`, and `R_RESEED`).
   - Each request results in a single `XOF` query.
   - All three requests allow an optional input AI, which can be adaptivly chosen by the attacker.

   Our results also apply to an attacker with $R' < R$ requests. Trivially, any attacker which wins the game using $R' < R$ requests can be converted into one that wins the game using exactly R requests, ignoring the final $R - R'$ responses.

4. In some situations, the attacker may realistically be able to *compromise* the DRBG state, i.e., to learn its value. Thus, after every request, the attacker is given the option to do so. The algorithm maintains a flag `corrupt`. The flag is set when the DRBG 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.

   A stronger attack model might allow the attacker to *set* the DRBG 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 8.4.

5. In parallel, the attacker is also allowed up to Q queries to the `XOF`. Note that neither party learns anything about the other's queries.

6. 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

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 statesize 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

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

Note that the result incentivizes to choose $H_{\text{rsd}} < H_{\text{init}}$, more precisely, to choose $H_{\text{rsd}} \approx H_{\text{init}} - \log_2(R)$.

For the proof, we first define which `XOF` queries made by the challenger are *uncorrupted*:

1. A query made to answer an instantiate or a reseed query is *uncorrupted*.
2. A query made when `corrupt` = `false` is *uncorrupted*.

In other words, a query is uncorrupted, if and only is made to answer a generate query *and* the flag `corrupt` is set.

Next, we now define certain *bad events*, which probabilities we will bound.

Bad1 Some of the challenger's DRBG states during the attack game repeat. I.e., there are $i \neq j$ with $V_i = V_j$, where V_i is the DRBG state after completing the attacker's ith request.

Bad2 The attacker makes a `XOF`-query, which matches the input from one of the uncorrupted queries the challenger made.

Finally, for the proof we will argue that if neither bad event occurs, one cannot distinguish $b = 0$ from $b = 1$. Thus, the advantage in distinguishing $b = 0$ from $b = 1$ is bounded by the probability of those bad events.

5.2 The Proof

Recall that the challenger's R states are denoted $V_1, \ldots V_R$. Write $U_1, \ldots, U_R$ for the corresponding `XOF` inputs, i.e., $V_i = \texttt{XOF}(U_i, |V|)$. Write $U^{\text{safe}} \subseteq \{U_1, \ldots, U_R\}$ for the subset of uncorrupted `XOF` inputs.

To prove the main result, we need several lemmas. The first one is almost trivial.

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

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

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 $2^{-\ell}$. □

The next lemma deals with Pr[Bad1], i.e., with the probability of the challenger to run into two colliding states $V_i = V_j$ when responding to the attacker's requests.

Lemma 2.

$$\Pr[\exists i, j : 1 \leq i < j \leq R, V_i = V_j] = \Pr[\text{Bad1}] \leq \frac{R^2}{2} * \left(\frac{1}{2^{H_{init}}} + \frac{1}{2^{|V|}} \right)$$

Proof. Firstly observe that if neither the i-th nor the j-th request are instantiate, then the first $|L|$ input bits to the `XOF` calls are V_{i-1} and V_{j-1}. (Note that $j > i > 1$, since the 1st request is instantiate.) If $V_{i-1} = V_{j-1}$, we already got a collision. If $V_{i-1} \neq V_{j-1}$, then $\Pr[V_i = V_j] = 1/2^{|V|}$, cf. lemma 1.

Secondly, if at least one of the i-th and the j-th request is instantiate, the corresponding `XOF` input is chosen from a probability distribution with min-entropy H_{init}. Thus, the

probability for the inputs to `XOF` calls to answer the i-th and the j-th request to collide is at most $2^{-H_{\text{init}}}$. If the inputs collide, then so do the outputs $V_i = V_j$. If the inputs do not collide, we apply lemma 1 and conclude $\Pr[V_i = V_j] = 1/2^{|V|}$.

For every pair (i, j) in the given range, $\Pr[V_i = V_j | 1 \leq i \leq j, (i > 1 \wedge V_{i-1} \neq V_{j-1})] \leq \frac{1}{2^{H_{\text{init}}}} + \frac{1}{2^{|V|}}$ holds. All in all, we get

$$\Pr[\text{Bad1}] \leq \binom{R}{2} * \left(\frac{1}{2^{H_{\text{init}}}} + \frac{1}{2^{|V|}}\right) \leq \frac{R^2}{2} * \left(\frac{1}{2^{H_{\text{init}}}} + \frac{1}{2^{|V|}}\right)$$

□

To complete the proof, we still have to deal with Pr[Bad2], i.e., with the probability of two `XOF` calls – one from the attacker and an uncorrupted one from the challenger – to have identical inputs.

Lemma 3.

$$\Pr[\text{Bad2} | \overline{\text{Bad1}}] \leq Q * \left(\frac{R}{2^{H_{init}}} + \frac{1}{2^{H_{rsd}}}\right)$$

Proof. Recall that we write $V_1, \ldots V_R$ for the challengers states. Thanks to $\overline{\text{Bad1}}$, all those states are disjoint. Recall that we write $U_1, \ldots, U_R$ for the corresponding inputs. Write $W_1, \ldots, W_Q$ for the inputs for the attacker's `XOF` queries. I.e., The event Bad2 is equivalent to

$$\exists U_i \in U^{\text{safe}}, j \in \{1, \ldots, Q\} : U_i = W_j.$$

In other words, we consider an attacker trying to guess one of the $U_i \in U^{\text{safe}}$. Write R_{inst}, R_{rsd}, and R_{gen} for the number of instantiate, reseed and generate requests, respectively. Note that $R = R_{\text{inst}} + R_{\text{rsd}} + R_{\text{gen}}$. Distinguish three cases:

1. The i-th request is instantiate. In such case, U_i begins with a seed, which is randomly chosen from $D_i^{H_{\text{init}}}$. Its min-entropy is H_{init}, thus $\Pr[W_j = U_i] \leq 2^{H_{\text{init}}}$. The probability for W_j to collide with any of those U_i is

$$\leq \frac{R_{\text{inst}}}{2^{H_{\text{init}}}}. \tag{2}$$

2. The i-th request is reseed. Now we consider two subcases. Observe that U_i begins with the previous state V_{i-1}, followed by the seed S_i.

   If V_{i-1} is uncorrupted, the probability to guess V_{i-1} is $1/2^{|V|}$. The probability to guess any of the uncorrupted V_{i-1} is

$$\leq \frac{R_{\text{rsd}}}{2^{|V|}} \leq \frac{R_{\text{rsd}}}{2^{H_{\text{init}}}}, \tag{3}$$

   since $H_{\text{init}} \leq |V|$.

   If V_{i-1} is corrupted, the attacker still has to guess the seed S_i, which is chosen from $D_i^{H_{\text{rsd}}}$. Thus, the probability for the attacker to guess $U_i = (V_{i-1} \parallel S_i \parallel \ldots)$ is at most $1/2^{H_{\text{rsd}}}$. Since $V_{i-1} \neq V_{j-1}$, the first $|V|$ bit of U_i uniquely determine S_i, hence the probability to guess any of the U_i is still

$$\leq \frac{1}{2^{H_{\text{rsd}}}}. \tag{4}$$

   In either case, the probability to guess the value U_i from a reseed request is at most

$$\leq \frac{R_{\text{rsd}}}{2^{H_{\text{init}}}} + \frac{1}{2^{H_{\text{rsd}}}}.$$

3. The i-th request is generate. Since U_i is uncorrupted, the attacker has to guess the first $|V|$ bit V_{i-1} of U_i Thus, $\Pr[W_j = U_i] \leq 1/2^{|V|} \leq 1/2^{H_{\text{init}}}$, since $H_{\text{init}} \leq |V|$. The probability to guess any of the U_i is

$$\leq \frac{R_{\text{gen}}}{2^{H_{\text{init}}}}.$$

In total, the probability to guess any of the uncorrupted U_i from either request,

$$\leq \frac{R_{\text{inst}}}{2^{H_{\text{init}}}} + \frac{R_{\text{rsd}}}{2^{H_{\text{init}}}} + \frac{1}{2^{H_{\text{rsd}}}} + \frac{R_{\text{gen}}}{2^{H_{\text{init}}}} \leq \frac{R}{2^{H_{\text{init}}}} + \frac{1}{2^{H_{\text{rsd}}}}.$$

The attacker makes Q queries, thus

$$\Pr[\text{Bad2}|\overline{\text{Bad1}}] \leq Q * \left(\frac{R}{2^{H_{\text{init}}}} + \frac{1}{2^{H_{\text{rsd}}}}\right).$$

□

The last lemma we need for the main result is about the adversarial advantage when no bad event occurs.

Lemma 4.

$$\Pr[\hat{b} = b \mid \overline{\text{Bad1}} \wedge \overline{\text{Bad2}}] = \frac{1}{2}$$

Proof. By the definition of the attack game (cf. alg. 3), all the outputs sent to the attacker which depend on b, are from uncorrupted generate requests.

The event $\overline{\text{Bad1}}$ implies that, since all the challenger's states V_i are different, all the `XOF` inputs U_i to answer generate requests are different. Thus, all the output bits generated in line 15 of the attack game stem from calls $\texttt{XOF}(U_i, \ldots)$ with different inputs U_i. Depending on b, an uncorrupted 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 $\texttt{XOF}(U_i, \ldots)$: if $b = 0$ it will get the same bits again, if $b = 1$ it will get independent random bits. □

Proof of theorem 1. According to lemma 4, $\Pr[\hat{b} = b|\overline{\text{Bad1}} \wedge \overline{\text{Bad2}}] = \frac{1}{2}$. I.e., without either bad event, the attacker has no information at all about b. Thus

$$\Pr[\hat{b} = b] \leq \frac{1}{2} + \Pr[\text{Bad1} \vee \text{Bad2}] \leq \frac{1}{2} + \Pr[\text{Bad1}] + \Pr[\text{Bad2}|\overline{\text{Bad1}}].$$

The result follows from combining the bounds from lemma 2 and lemma 3. □

5.3 Matching Attacks: the Main Result is Tight

In this section, we will sketch attacks which closely match our claimed security bounds. We assume each D_i^h to be the uniform distribution of h-bit values. This tightly matches the claimed min-entropy h for the D_i^h.

Attack 1 ($R \approx 2 * 2^{H_{\text{init}}/2}$): The core idea for this attack is to maximise the probability of the event Bad1 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_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.

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_i for reseed. The input state V_{i+1} for generate is computed by $V_{i+1} = \texttt{XOF}(V_i \parallel S_i, |V|)$ from the unknown seed $S_i \leftarrow\$\, D_i^{H_{\text{rsd}}}$.

If $b = 0$, the visible output consists of the ℓ rightmost bits from $\texttt{XOF}(V_{i+1}, |V| + \ell)$. By trying out all $2^{H_{\text{rsd}}}$ choices for S_i, the attacker can find V_{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 $\texttt{XOF}(\texttt{XOF}(V_i \parallel S, |V|), |V| + \ell)$ match ℓ random bits is negligible.

5.4 Quantum Security

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. While a classical implementation of attack 2a or 2b when 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. I.e., if we assume the attacker is not willing to wait for more than the wall-clock time 2^{70} cryptographic operations would take,[6] either on a classical or on a quantum computer, then 100-bit classical security is at least as good as 85-bit quantum security. In general, $70 + 2t$ bits of classical security are at least as good as $70 + t$ bit of quantum security.[7]

5.5 Consequences of Main Result

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 alogorithm. E.g., with $R = 2^{64}$, $H_{\text{init}} = 192$, 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 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}	$\log_2(R)$	H_{rsd}	security level	
			classical	quantum
192	64	128	128	64
220	50	170	170	85
240	48	192	192	96
320	64	256	256	128

6 Using the AI for Personalization

Consider a device i, handling R_i requests. The security of device i seems to be bounded by our main result, for $R = R_i$. But consider the existence of lots of devices – maybe millions – all using the same XDRBG. The user running device i has no control – and perhaps not even knowledge – about the number of requests handled by other users' devices.

The attacker is still constrained by our main result. But if the attacker is happy to break any device, regardless of which one, the value R is now $R = R_\Sigma = \sum_i R_i$. If R_Σ exceeds a given upper bound on R, then a security claim made on the basis of this bound is void.[8] Also, consider user i being of only moderate interest for a given attacker. The attacker would not spend millions of dollars to attack user i's XDRBG. But assume user j to run the same XDRBG, and, for whatever reason, the attacker actually spends millions of dollars to attack user j. If so, user i might find himself or herself in the sad situation of becoming some *by-catch* for the attacker.

The underlying problem is that the attack is essentially about finding matching XOF queries, and users i and j (and many others) are using the same XOF. To avoid this issue, it suffices to make sure that the XOF inputs from running i's XDRBG are always different

[6] 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 size 2^{70} of the threshold is, of course, open for debate.

[7] 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.

[8] In practice, one can still question any attacker's ability to observe millions of devices. Also, as indicated by our main result, the security claims do not simply collapse, but degrade gracefully with growing R. Nevertheless, exceeding the bound on R deserves to be taken seriously.

from the XOF inputs j makes when running the XDRBG. I.e., i and j (and all the others) can prevent this by personalizing their XDRBGs.

Personalization for implementation i works as follows:

- Choose a unique identification string $\langle \text{name} \rangle_i$.
- For for all requests, use $\text{AI}_{\text{step}} = \langle \text{name} \rangle_i$ as the additional input.

Essentially, $\langle \text{name} \rangle_i$ serves as an implementation-wide constant. As long as $\langle \text{name} \rangle_i \neq \langle \text{name} \rangle_j$ for two devices $j \neq j$, the security of device i is not affected by attacks on device j, and vice versa.[9]

An alternative approach, with similar benefits, is randomization. Assume a fixed number $\ell_{\text{id}} \geq 64$ for the size of a random identifier. For device i, do the following:

1. Request instantiate with AI_1 being the empty string.
(The first step is instantiate, anyway.)

2. Request to generate ℓ_{rnd} bits of output, with AI_2 being the empty string.
Refer to the result as $\langle \text{rnd} \rangle_i \in \{0,1\}^{\ell_{\text{rnd}}}$.

3., 4., ... For step ≥ 3: use $\text{AI}_{\text{step}} = \langle \text{rnd} \rangle_i$ as the additional input.

Note that we do not require the secrecy of $\langle \text{rnd} \rangle_i$. I.e., as long as $\langle \text{rnd} \rangle_i \neq \langle \text{rnd} \rangle_j$, users enjoy the same benefits as when using $\langle \text{name} \rangle_i \neq \langle \text{name} \rangle_j$ – even when the DRBG state has been compromised after the first request.

In almost all practical cases, $|V|$ will be so large that $|V| \gg R_\Sigma^2$ can always be taken for granted, even given millions if devices using the same XDRBG. In this case, it suffices to personalize or randomize only instantiate requests, while the value AI for reseed and generate can just be the empty string. The reason is as follows: Firstly, due to $|V| \gg R_\Sigma^2$, we can neglect the probability of the event Bad1, i.e., the probability of some of the challenger's state during the attack game to repeat. Secondly, observe that the probability of Bad2 is the probability of an attacker to guess the input of any of the challenger's uncorrupted XOF queries. Instantiate request do not use the old state as a part of their input. So if the joint number of all instantiate requests from all devices is $R_{\Sigma\text{init}}$, then the probability to guess any of them with Q queries can be close to $R_{\Sigma\text{init}} Q / 2^{H_{\text{init}}}$. On the other hand, reseed and generate requests include the old state as a part of their input. The probability to guess any of those inputs is thus less than $R_\Sigma Q / |V|$, i.e., negligible.

We recommend to personalize or randomize all implementations of the XDRBG which require a bound $R = 2^{50}$ or smaller for their security claims. This could be done by feeding the personalization string $\langle \text{name} \rangle$ or $\langle \text{rnd} \rangle$ into the AI-input of all instantiate requests. Alternatively, if the XOF provides a personalization option, one can make use of that option and leave the AI 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.

[9] As a side effect, personalization can cover some issues related to flawed entropy estimates. In the past, some random generators did fail to provide the expected entropy early in the attack game. Thus, different users might actually sample the same seed. [23] used this to attack RSA keys, which shared one of the two primes – presumably the one generated first. Assume the XDRBGs from user i and user j that happen to sample the same seed at some point of time. If their additional inputs are the also same (e.g., empty), then the attacker will simply observe identical outputs from subsequent generate requests. But if the additional inputs supplied by i and j to the XDRBG are different, the attacker will only observe seemingly independent pseudorandom output strings. This seems to be beneficial for the user, though we also have to point out that an attacker might still discover and exploit the flawed entropy estimates, and personalization also makes it more difficult for the security engineer to discover such an issue.

7 Concrete Proposals

7.1 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 bit of classical security and 64 bit of quantum security. This matches category one (the lowest category) from the NIST post-quantum security criteria, see Appendix C. 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, 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 appoximate security levels. The first assumes SHAKE128 with its 256-bit capacity, the second and third SHAKE256 with its 512-bit capacity. We set $|V| =$ capacity. The *category* refers to the NIST post-quantum criteria, cf. Appendix C.

	capacity	H_{init}	$\log_2(R)$	H_{rsd}	promised security		
					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 $= 2048$ bits on the maximum output length from one single generate call. That is, each GENERATE call must return no more than 2048 bits (256 bytes) of output. To generate X bits 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, $k \in \{128, 192, 256\}$. The min-entropy needed for instantiation is defined in terms of the security level: $H_{\text{init}} \geq 3k/2, H_{\text{rsd}} \geq k$. 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 AIS20/31 [32], (still a draft as of this writing), which defines lower-limits on effective internal state size and entropy provided to the DRNG[10]. The current draft of AIS 20/31 requires 240 bits of Shannon entropy[11] for instantiation, so $H_{\text{init}} \geq 240, H_{\text{rsd}} \geq 240$. 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.

7.2 Alternative Proposals Based on the ASCON Permutation

Recently, [30], NIST has announced plans to standardize ASCON [13], a lightweight family of authenticated encryption and hashing algorithms based on a 320-bit permutation. We anticipate corresponding standards for ASCON-based XOFs.

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

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

[11] Shannon entropy of a random variable is never greater than its min-entropy, so requiring 240 bits min-entropy will always suffice to meet AIS 20/31 requirements.

only provide $c/2$ bit of classical and $c/3$ bit 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 bit 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 bit of classical and 96 bit 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 remaining two are the attempt to provide improved security, especially quantum security, using the same 320-bit permutation. XDRBG-L-128-85 improves the quantum security by restricting the number of requests to 2^{50}, instead of 2^{64}. We thus recommend every device running XDRBG-L-128-85 to personalize the XDRBG, as described in Section 6. 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.

Table 3: Three proposals for lightweight DRBG standards and their approximate security. We set $|V| =$ capacity. The *category* column refers to the NIST post-quantum criteria, cf. Appendix C. 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}	$\log_2(R)$	H_{rsd}	promised security		
					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)

7.3 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 $\rho = r - |V|$ bit 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 $|\text{AI}|$ needs to make $\lceil (|V| + \sigma + |\text{AI}|)/r \rceil$ 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

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

- 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

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

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

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

For SHAKE128 (i.e., for XDRBG-128), $\sigma = 1088$, and for SHAKE256 (i.e., for XDRBG-192 and XDRBG-256), $\sigma = 576$, i.e., $|V| \leq r$. For the XDRBG instantiations based on the ASCON permutation, we have either rate $r = 64$ or $r = 32$.

As a concrete (and, in our opinion, quite typical) example, consider $\sigma = 512$, $|ai| = 64$, and $\ell = 256$. Then each of XDRBG-128, XDRBG-192 and XDRBG-256 only needs to make one permutation call for each request. The rate and state size of XDRBG-L-128 and XDRBG-L-128-85 are $r = 64$ and $|V| = 256$, so each request requires 12 to 16 permutation calls. For XDRBG-L-144-96 the rate is $r = 32$ and the state size is $|V| = 288$. This implies 28 to 32 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 [21, 9], a reseed *must* wait until sufficient entropy is available to avoid the iterative guessing attack / premature next condition.

Also, one can introduce some form of a *conditional reseed*: **if** H_{rsd} bit of min-entropy are available, **then** reseed now **else** continue without reseeding. This approach may deserve consideration for future research. Note that there is a potential side-channel vulnerability: The attacker might observe if, whenever the DRBG executes a conditional reseed, the DRBG actually reseeds or not. If so, this could actually cause a loss of entropy for the DRBG: up to one bit of Shannon-entropy each time the conditional reseed is called.

8.3 Padding the Input for the `XOF` Queries

We considered adding padding, to guarantee that `XOF` queries resulting from different DRBG calls would be distinct, even if the state or the seed accidentally happened to be the same for those queries. We also considered adding a length field, to ensure a one to one correspondence between parameters to the DRBG call and inputs to `XOF`.

This option would not weaken the security bound we have proven. Neither would it significantly strengthen that bound: the bad events are the same, and their probabilities will be the same, or marginally smaller. While adding these would be a form of prudent security engineering, we decided against it, because its benefits would not justify the additional costs for some likely future instantances of `XDRBG`.

8.4 Stronger Attack Model

A variation of the attack game could allow the attacker to compromise the DRBG state by *setting or overwriting* it, 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 ever overwrite the DRBG state by a chosen or known value V^*. But for readers who need to provide a defence even against 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 a known constant, the 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 AIS20/31 sets $H_{\text{init}} = H_{\text{rsd}}$.

Regarding the first recomendation, 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 1 by

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

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

9 Conclusions

Drawing on previous work in [8, 10], 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 SP800-90 and AIS20/31, 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.

9.1 Open Questions

The security analysis in this paper focuses on security against classical attacks, with non-adversarial entropy sources. This leaves two interesting directions for future research:

1. Our analysis for quantum security focused on attacks, namely on the application of Grover's algorithm. Also, recall the cautionary note from Footnote 4 regarding the quantum security of sponge-based XOFs. Neverthless, 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 [36]. 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 SP800–90 or AIS20/31), it would be interesting to know the security bounds for oracle-dependent sources, using the techniques of [10]. 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., 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.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Lenstra, A. K., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., and Wachter, C. Public keys. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*, Springer, pp. 626–642.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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 Onput

`XDRBG`, like the DRBGs in SP800–90A, allows for an optional additional input to each DRBG call. This input maybe 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. In both INSTANTIATE and RESEED, the additional input is simply appended to the end of the seed before the `XOF` query is made. Trivially, as long as the contents of the seed are independent of the contents of the additional input, the string input to the `XOF` must thus have $h_1 + h_2$ bits of min-entropy. Thus, in the case of lower than expected entropy in the seed, providing sufficient entropy the additional input can still ensure a secure instantiation or reseed.

B 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).

C NIST security categories for post-quantum cryptography

In [29], NIST defines five security categories for submissions to the Post-Quantum Cryptography process. 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