An Analysis of the NIST SP 800-90A Standard

Joanne Woodage[1], Dan Shumow[2]

[1]Royal Holloway, University of London
[2] Microsoft Research

Abstract. We investigate the security properties of the three deterministic random bit generator (DRBG) mechanisms in the NIST SP 800-90A standard [2]. This standard received a considerable amount of negative attention, due to the controversy surrounding the now retracted DualEC-DRBG, which was included in earlier versions. Perhaps because of the attention paid to the DualEC, the other algorithms in the standard have received surprisingly patchy analysis to date, despite widespread deployment. This paper addresses a number of these gaps in analysis, with a particular focus on HASH-DRBG and HMAC-DRBG. We uncover a mix of positive and less positive results. On the positive side, we prove (with a caveat) the robustness [16] of HASH-DRBG and HMAC-DRBG in the random oracle model (ROM). Regarding the caveat, we show that if an optional input is omitted, then – contrary to claims in the standard — HMAC-DRBG does not even achieve the (weaker) property of forward security. We also conduct a more informal and practice-oriented exploration of flexibility in implementation choices permitted by the standard. Specifically, we argue that these DRBGs have the property that partial state leakage may lead security to break down in unexpected ways. We highlight implementation choices allowed by the overly flexible standard that exacerbate both the likelihood, and impact, of such attacks. While our attacks are theoretical, an analysis of two open source implementations of CTR-DRBG shows that potentially problematic implementation choices are made in the real world.

1 Introduction

Secure pseudorandom number generators (PRNGs) underpin the vast majority of cryptographic applications. From generating keys, nonces, and IVs, to producing random numbers for challenge responses, the discipline of cryptography — and hence system security — critically relies on these primitives. However, it has been well-established by a growing list of real-world failures [43], [36], [21], [8], that when a PRNG is broken, the security of the reliant application often crumbles with it. Indeed, with much currently deployed cryptography being effectively 'unbreakable' when correctly implemented, exploiting a weakness in the underlying PRNG emerges as a highly attractive target for an attacker. As such, it is of paramount importance that standardized PRNGs are as secure as possible.

The NIST Special Publication 800-90A Recommendation for Random Number Generation Using Deterministic Random Bit Generators (NIST SP 800-90A) [2] has had a troubled history. The first version of this standard included the now infamous DualEC-DRBG, which was long suspected to contain a backdoor inserted by the NSA [40]. This suspicion was reportedly confirmed by documents included in the Snowden leaks [33], leading to a revision of the document that removed the disgraced algorithm.

Perhaps because of the focus on DualEC-DRBG, the other algorithms standardized in the document have received surprisingly little attention and analysis. These PRNGs — which respectively use a hash function, HMAC, and a block cipher as their basic building blocks — are widely used. Indeed, these are the *only* approved PRNGs for cryptographic software or hardware seeking FIPS certification [19, 42]. While aspects of these constructions have been analyzed [11, 23, 22, 39, 24, 37] and some implementation considerations discussed [7], these works tend to make significant simplifying assumptions and / or treat only certain algorithms rather than the constructions as a whole. There has not to date been a deeper analysis of these standardized DRBGs, either investigating the stronger security properties claimed in the standard or taking into account the (considerable) flexibility in their specification.

The constructions provided in the NIST SP 800-90A are somewhat nonstandard. Even the term DRBG is rare, if not absent from the literature, which favors the term PRNG. Similarly the NIST

DRBGs — which return variable (and sizable) length outputs upon request, and support a variety of optional inputs and parameters — do not fit cleanly into the usual security models for PRNGs. With only a limited amount of formal analysis in the literature to date, coupled with the fact that the standardization of these algorithms did not follow from a competition or widely publicly vetted process, this leaves pseudorandom number generation in large parts of software relying on relatively unanalyzed algorithms.

In particular, the standard claims that each of the NIST DRBGs is 'backtracking resistant' and 'prediction resistant'. The former security property is the familiar forward security notion for PRNGs first formalized by Bellare et al. [5], which guarantees that in the event of a state compromise output produced prior to the point of compromise remains secure. The latter property ensures that if the state is compromised and subsequently reseeded with sufficient entropy then security will be recovered. Somewhat surprisingly, to the best of our knowledge neither of these properties have been formally investigated and proved. In fact, the NIST DRBG algorithms which are responsible for initial state generation and reseeding do not seem to have been analyzed at all in prior work.

The goal of this paper is to address some of these gaps in analysis.

1.1 Contributions

We conduct an investigation into the security of the NIST SP 800-90A DRBGs, with a focus on HASH-DRBG and HMAC-DRBG. We pay particular attention to flexibilities in the specification of these algorithms, which are frequently abstracted away in previous analysis. We set out to analyze the algorithms as they are specified and used, and so sometimes make heuristic assumptions in our modeling (namely, working in the random oracle model (ROM) and assuming an oracle-independent entropy source). We felt this more constructive than modifying the constructions solely to derive a proof under weaker assumptions, and explain the rationale behind all such decisions.

Robustness proofs. The notion of robustness, introduced by Dodis et al. [16], captures both backtracking and prediction resistance and is the 'gold-standard' for PRNG security. For our main technical results, we analyze HASH-DRBG and HMAC-DRBG within this framework. As a (somewhat surprising) negative result, we show that implementations of HMAC-DRBG for which optional strings of additional input are not always included in next calls (see Section 3) are not forward secure. This is contrary to claims in the standard that the NIST DRBGs are backtracking resistant. This highlights the importance of formally proving security claims which at first sight may seem obviously correct, and of paying attention to implementation choices.

As positive results, we prove that HASH-DRBG and HMAC-DRBG (called with additional input) are robust in the ROM. The first result is fully general, while the latter is with respect to a class of entropy sources which includes those approved by the standard.

A key challenge is that the NIST DRBGs do not appear to have been designed with a security proof in mind. As such, seemingly innocuous design decisions turn out to significantly complicate matters. The first step is to reformulate robustness for the ROM. Our modeling is inspired by Gazi and Tessaro's treatment of robustness in the ideal permutation model [20]. We must make various adaptations to accommodate the somewhat unorthodox NIST DRBGs, and specifiying the model requires some care. It is for this reason that we focus on HASH-DRBG and HMAC-DRBG in this work, since they map naturally into the same framework. Providing a similar treatment for CTR-DRBG would require different techniques, and is an important direction for future work.

At first glance, it may seem obvious that a PRNG built from a random oracle will produce random looking bits. However, formally proving that the constructions survive the strong forms of compromise required to be robust is far from trivial. While the proofs employ fairly standard techniques, certain design features of the algorithms introduce unexpected complexities and some surprisingly fiddly analysis. Throughout this process, we highlight points at which a minor design modification would have allowed for a simpler proof.

Implementation flexibilities. We counter these formal and (largely) positive results by offering a more informal discussion of flexibilities in the standard. We argue that when the NIST DRBGs are used to produce many blocks of output per request — a desirable implementation choice in terms of efficiency, and permitted by the standard — then the usual security models may overlook

important attack vectors against these algorithms. Taking a closer look, we propose an informal security model in which we suppose an attacker compromises part of the state of the DRBG — for example through a side-channel attack — *during* an output generation request. Reconsidered within this framework, we find that each of the constructions admits vulnerabilities which allow an attacker to recover unseen output. We find a further flaw in a certain variant of CTR-DRBG which allows an attacker who compromises the state to also recover strings of additional input — which may contain secrets — previously fed to the DRBG. While our attacks are theoretical in nature, we follow this up with an analysis of the open-source OpenSSL and mbed TLS CTR-DRBG implementations, which shows that the implementation decisions we highlight as potentially problematic are taken by implementors in the real world. We conclude with a number of reflections and recommendations for the safe use of these DRBGs.

Related work. A handful of prior works have analyzed the NIST DRBGs as deterministic pseudorandom generators (PRGs). That is to say, they prove that the output generation algorithm of the DRBG produces pseudorandom bits when applied to an *ideally random* initial state e.g., $S_0 = (K_0, V_0, cnt_0)$ for uniformly random K_0, V_0 in the case of CTR-DRBG and HMAC-DRBG. This is a substantial simplification; in the real world, these state components must be derived from the entropy source using the setup algorithm. Campagna [11] and Shrimpton and Terashima [39] provide such a treatment of CTR-DRBG, while Hirose [22] and Ye et al. [24] give proofs for HMAC-DRBG. This latter work also provides a formal verification of the mbedTLS implementation of HMAC-DRBG. None of these works model initial state generation or reseeding; as far as we are aware, ours is the first work to analyze these algorithms for HASH-DRBG and HMAC-DRBG. With the exception of [22], they do not model the use of additional input. Moreover, pseudorandomness of output is a much weaker security model than robustness and does not allow any form of state compromise. Kan [23] considers the assumptions underlying the security claims of the DRBGs. To our knowledge, this is the only previous work to consider HASH-DRBG. However, the analysis is rather informal and non-standard.

In [37], Ruhault claims a potential attack against the robustness of CTR-DRBG. However, the specification of the BCC function in that work (a CBC-MAC-like function which is used by the CTR-DRBG_df algorithm) is different to that provided by the standard. Namely, in [37] BCC is defined to split the input $IV \,\|\, S$ into n 128-bit blocks ordered from right to left as $[B_n, \ldots, B_1]$. However, in the standard these blocks are ordered left to right $[B_1, \ldots, B_n]$. This leads to the blocks being processed in a different order by BCC. The attack from [37] does not work when the correct BCC function is used, and does not seem possible to fix.

2 Preliminaries

Notation. The set of binary strings of length n is denoted $\{0,1\}^n$. We write $\{0,1\}^*$ to denote the set of all binary strings, and $\{0,1\}^{\leq n}$ to denote the set of binary strings of length at most n-bits; we include the empty string ε in both sets. We convert binary strings to integers, and vice versa, in the standard way. We let $x \oplus y$ denote the exclusive-or (XOR) of two strings $x, y \in \{0,1\}^n$, and write $x||y$ to denote the concatenation of two strings x and y. We write $\mathrm{left}(x, \beta)$ (resp. $\mathrm{right}(x, \beta)$) to denote the leftmost (resp. rightmost) β bits of string x, and $\mathrm{select}(x, \alpha, \beta)$ to denote the substring of x consisting of bits α to β inclusive. We let $[j_1, j_2]$ denote the set of integers between j_1 and j_2 inclusive. For an integer $j \in \mathbb{N}$, we write $(j)_c$ to represent j encoded as a c-bit binary string. The notation $x \xleftarrow{\$} \mathcal{X}$ denotes sampling an element uniformly at random from the set $\mathcal{X}$. We let $\mathbb{N} = \{1, 2, \ldots\}$ denote the set of natural numbers, and let $\mathbb{N}^{\leq n} = \{1, 2, \ldots, n\}$.

Entropy and Cryptographic Components. In Appendix A we recall the standard definitions of worst-case and average-case min-entropy, along with the usual definitions of pseudorandom functions (PRFs) and block ciphers.

Pseudorandom number generators with input. A *pseudorandom number generator with input* (PRNG) [16] produces pseudorandom bits and offers strong security guarantees (see Section 5) when given continual access to an imperfect source of randomness. We define PRNGs formally below, and then discuss our choice of syntax.

Definition 1. *A pseudorandom number generator with input (PRNG) is a tuple of algorithms* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$*, with associated parameter set* $(\alpha_{min}, \alpha_{max}, \alpha_{add}, \alpha_{out})$*, defined as follows:*

- setup : $\mathsf{salt} \times \cup_{i=\alpha_{min}}^{\alpha_{max}}(\{0,1\}^i) \times \mathcal{N} \to \mathcal{S}$ *takes as input a salt* $X \in \mathsf{salt}$ *where* salt *denotes the salt space of the PRNG, an entropy sample* $I \in \cup_{i=\alpha_{min}}^{\alpha_{max}}\{0,1\}^i$*, and a nonce* $N \in \mathcal{N}$ *where* $\mathcal{N}$ *denotes the nonce space of the PRNG, and returns an initial state* $S_0 \in \mathcal{S}$*, where* $\mathcal{S}$ *denotes the state space of the PRNG.*
- $\mathsf{refresh}$: $\mathsf{salt} \times \mathcal{S} \times \cup_{i=\alpha_{min}}^{\alpha_{max}}(\{0,1\}^i) \to \mathcal{S}$ *takes as input a salt* $X \in \mathsf{salt}$*, a state* $S \in \mathcal{S}$*, and an entropy sample* $I \in \cup_{i=\alpha_{min}}^{\alpha_{max}}\{0,1\}^i$*, and returns a state* $S' \in \mathcal{S}$*.*
- next : $\mathsf{salt} \times \mathcal{S} \times \mathbb{N}^{\leq \alpha_{out}} \times \{0,1\}^{\leq \alpha_{add}} \to \{0,1\}^{\leq \alpha_{out}} \times \mathcal{S}$ *takes as input a salt* $X \in \mathsf{salt}$*, a state* $S \in \mathcal{S}$*, a parameter* $\beta \in \mathbb{N}^{\leq \alpha_{out}}$*, and a string of additional input* $addin \in \{0,1\}^{\leq \alpha_{add}}$*, and returns an output* $R \in \{0,1\}^\beta$*, and an updated state* $S' \in \mathcal{S}$*.*

If a PRNG always has $X = \varepsilon$ or $addin = \varepsilon$ (indicating that, respectively, a salt or additional input is never used), then we omit these parameters.

Discussion. Our definition follows that of Dodis et al. [16], with a number of modifications. The key differences are: **(1)** we extend the PRNG syntax to accommodate additional input, nonces and a parameter indicating the number of output bits requested, all of which are part of the NIST DRBG interface; **(2)** following Shrimpton et al. [38], we define setup to be the algorithm which constructs the initial state of the PRNG from a sample drawn from the entropy source, and assume that the salt X is generated externally and supplied to the PRNG; and **(3)** we allow entropy samples and outputs to take any length in a range indicated by the parameters of the PRNG, rather than being of fixed length. We provide a full discussion of these modifications in Appendix A. The SP 800-90A standard uses the term *deterministic random bit generator* (DRBG) instead of the more familiar PRNG. We use these terms interchangeably.

3 The NIST SP 800-90A Standard

3.1 Overview of the Standard

The standard defines three DRBG mechanisms, HASH-DRBG, HMAC-DRBG, and CTR-DRBG, based on a hash function, HMAC, and a block cipher respectively.

Algorithms and mapping into Definition 1. The standard specifies a tuple of $(\mathsf{Instantiate}, \mathsf{Reseed}, \mathsf{Generate})$ algorithms for each of the DRBGs. These algorithms map directly into the $(\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ algorithms in the PRNG model of Definition 1. For consistency with the usual PRNG syntax, we refer to the NIST DRBG algorithms as $(\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ throughout. The NIST DRBGs are not specified to take a salt; as such when mapping into the syntax of Definition 1 we always take $X = \varepsilon$ and $\text{Seed} = \emptyset$, and omit these parameters from the subsequent definitions. We discuss the lack of a salt further in Section 5. The standard also defines an update algorithm for CTR-DRBG and HMAC-DRBG, and derivation functions for HASH-DRBG and CTR-DRBG, which are called as subroutines by the other algorithms. These algorithms are used to derive new state variables, and incorporate provided data into them. Additionally, the CTR-DRBG derivation function is used to pre-process entropy samples and additional inputs, typically into a string of shorter length, prior to their use by other DRBG algorithms. A pseudocode presentation of the component algorithms of each of the DRBGs is given in Figure 1[1], and is discussed in more detail in Section 4. A set of sample parameters for typical instantiations of the DRBGs is given in Appendix G.2. The full list of allowed instantiations is given in the standard.

DRBG functions. The setup, $\mathsf{refresh}$, and next algorithms underly (respectively) the `Instantiate`, `Reseed` and `Generate` functions of the DRBG. When called, these functions check the validity of the request (e.g., that the number of requested bits does not exceed α_{out}), and return an error if these checks fail. If not, the function fetches the internal state of the DRBG, along with any other inputs required by the algorithms (such as entropy inputs, a nonce, and so on), and the

[1] We do not directly analyze the $\mathsf{refresh}$ algorithm and derivation function for CTR-DRBG, and so defer the presentation of these algorithms to Appendix G.

underlying algorithm is applied to these inputs. The resulting outputs are returned to the caller and / or used to update the internal state, and the successful status of the call is indicated to the caller. Here we abstract away this process to avoid cluttering our exposition. To this end, we assume that all required inputs are provided to the algorithm in question (without modeling how these are fetched), and assume that all inputs and requests are valid, omitting the success / error notifications. We use calling a function and invoking the underlying algorithm interchangeably. A DRBG mechanism also includes an `Uninstantiate` function which erases the internal state and a `Health Test` function, which is used to test if the other functions in an implementation are performing correctly. We do not model these functions in this work.

The DRBG State. The standard defines the *working state* of a DRBG to be the set of stored variables which are used to produce pseudorandom output. The *internal state* is then defined to be the working state plus administrative information, which indicates the security strength of the instantiation and whether prediction resistance is supported (see below). We typically omit administrative information as this shall be clear from the context. By the 'state' of the DRBG (denoted S), we mean the working state unless otherwise specified.

Entropy sources and instantiation. The DRBGs must have access to an *approved entropy source*[2] during initial state generation via setup. The DRBG uses the function `Get_entropy_input` to request an entropy sample I of length within the range $[\alpha_{min}, \alpha_{max}]$ (see Definition 1), and containing a given amount of entropy (discussed further below). For all DRBG mechanisms except CTR-DRBG implemented without a derivation function, the `Instantiate` function must also acquire a nonce to be used during instantiation. Nonces must either contain $\gamma^*/2$-bits of min-entropy, or not be expected to repeat more than such a value would. Examples of suitable nonces given in the standard include strings drawn from the entropy source, time stamps, and sequence numbers. Once the initial state has been constructed, the DRBG is said to be *instantiated*. A DRBG implementation can support multiple simultaneous instantiations, which are differentiated between using state handles. Here we assume that each DRBG supports a single instantiation, and so omit handles.

Reseeding. If the DRBG has continual access to an entropy source, then the DRBG is said to support *prediction resistance*. In this case, entropy samples drawn from the source may be periodically incorporated into the DRBG state via refresh. We assume that a DRBG instantiation always supports prediction resistance, and omit the parameters indicating this from the state and function calls. Reseeds may be explicitly requested by the consuming application, or triggered by a request in a next call (in which case, a refresh is performed before the state is passed to next). Additionally, a DRBG instantiation specifies a parameter *reseed_interval*, which indicates the maximum number of output generation requests before a reseed is forced. For all allowed instantiations (with the exception of CTR-DRBG instantiated with 3-KeyTDEA), *reseed_interval* may be at most 2^{48}. The number of next calls since the last refresh is recorded by a state component called a reseed counter (*cnt*). In keeping with the usual modelling of PRNGs, we assume here that reseeds are always explicitly requested; this is without loss of generality.

Security strength. An instantiation of a NIST DRBG is parameterized by a *security strength* $\gamma^* \in \{112, 128, 192, 256\}$. The standard requires that all entropy samples used in initial state generation and reseeding contain at least γ^*-bits of entropy.[3]

Output generation. Outputs of varying lengths up to α_{out}-bits may be requested by the caller via the parameter β, which forms part of the input to the next function. For all allowed instantiations (with the exception of CTR-DRBG instantiated with 3-KeyTDEA), α_{out} may be as large as 2^{19}.

Additional input. The standard gives the option for strings of additional input (denoted *addin*) to be provided to the DRBG by the caller during next calls. These inputs may be public or predictable (e.g., device serial numbers and time stamps), or may contain secrets. If these inputs

[2] Either a live entropy source as approved in SP 800-90B [41], or a (truly) random bit generator as per SP 800-90C [3].

[3] In contrast, the notion of robustness for PRNGs [16] (see Section 5), requires that a PRNG is secure when reseeded with sets of entropy samples which *collectively* have γ^*-bits of entropy. Looking ahead to Section 6, we analyze HASH-DRBG with respect to this stronger notion.

do contain entropy, they may provide a buffer in the event of a system failure or compromise. Here, we assume that an instatiation is either always or never called with additional input during next calls. The standard also allows optional additional input to be included in refresh calls, and during setup (in the form of a *personalization string*). We omit these here for simplicity.

4 Algorithms

We now describe the component algorithms of the NIST DRBGs (Figure 1).

4.1 HASH-DRBG.

HASH-DRBG is built from an (unkeyed) cryptographic hash function $\mathsf{H} : \{0,1\}^{\leq\omega} \to \{0,1\}^{\ell}$. The working state is defined $S = (V, C, cnt)$, where the counter $V \in \{0,1\}^{L}$ and constant $C \in \{0,1\}^{L}$ are the security critical state variables. The standard does not explicitly state the role of C; however its purpose would appear to be preventing HASH-DRBG falling into a sequence of repeated states. We discuss this further in Appendix B.

Algorithms. Both the setup and refresh algorithms of HASH-DRBG derive a new state by applying the derivation function HASH-DRBG_df to the entropy input and (in the case of refresh) the previous counter. Output generation via next proceeds as follows. If additional input is used in the call, it is hashed and added into the counter V (lines 3 - 5). Output blocks are then produced by hashing the counter in CTR-mode (lines 7 - 10). At the conclusion of the call, the counter V is hashed with a distinct prefix prepended, and the resulting string — along with the constant C and reseed counter cnt — are added into V to update the counter (lines 12 - 13).

4.2 HMAC-DRBG

HMAC-DRBG is built from the function $\mathsf{HMAC} : \{0,1\}^{\ell} \times \{0,1\}^{\leq\omega} \to \{0,1\}^{\ell}$. The working state is of the form $S = (K, V, cnt)$, where the key $K \in \{0,1\}^{\ell}$ and counter $V \in \{0,1\}^{\ell}$ are the security critical state variables.

Algorithms. The setup and refresh algorithms of HMAC-DRBG both use the update subroutine to incorporate an entropy sample I into K and V. For setup, these variables are initialized to $K = 0\mathtt{x}00\ldots00$ and $V = 0\mathtt{x}01\ldots01$ prior to this process. The next algorithm for HMAC-DRBG proceeds as follows. If additional input is used, this is incorporated into K and V via the update function (lines 3 - 4). Output is then generated by iteratively computing $V \leftarrow \mathsf{HMAC}(K, V)$, and concatenating the resulting strings (lines 6 - 8). At the conclusion of the call, both key and counter are updated via the update function (line 10).

4.3 CTR-DRBG

CTR-DRBG is built from a block cipher $\mathsf{E} : \{0,1\}^{\kappa} \times \{0,1\}^{\ell} \to \{0,1\}^{\ell}$. The working state is defined $S = (K, V, cnt)$, where the key $K \in \{0,1\}^{\kappa}$ and counter $V \in \{0,1\}^{\ell}$ are the security critical state variables. Ideally, CTR-DRBG would be initialized with an ideally random state $S_0 = (K_0, V_0, cnt_0)$ where $K_0 \leftarrow\!\!\$\ \{0,1\}^{\kappa}$, $V_0 \leftarrow\!\!\$\ \{0,1\}^{\ell}$, and $cnt_0 = 1$. As discussed in Section 1, we do not analyze setup and refresh for CTR-DRBG; these algorithms are shown in Appendix G.

Algorithms. There are two variants of CTR-DRBG depending on whether a derivation function is used. A pseudocode description of this function is given in Appendix G. The next algorithm for CTR-DRBG proceeds as follows. First, any additional input is incorporated into the state via the update function (line 8). If a derivation function is used, the string of additional input is conditioned into a $(\kappa+\ell)$-bit string with the CTR-DRBG_df prior to this process (line 5). If a derivation function is not used, the additional input string is restricted to be at most $(\kappa + \ell)$-bits in length. Output blocks are then iteratively generated using the block cipher in CTR-mode (lines 11 - 13). At the conclusion of the call, both K and V are updated via an application of the update function (line 15).

HASH-DRBG_df
Require: $input\_string$, $(num\_bits)_{32}$
Ensure: req_bits
$temp \leftarrow \varepsilon$; $m \leftarrow \lceil num\_bits/\ell \rceil$
For $i = 1, \ldots, m$
  $temp \leftarrow temp \,\|\, \mathsf{H}((i)_8 \,\|\, (num\_bits)_{32} \,\|\, input\_string)$
$req\_bits \leftarrow \text{left}(temp, num\_bits)$
Return req_bits

HASH-DRBG setup
Require I, N
Ensure: $S_0 = (V_0, C_0, cnt_0)$
$seed\_material \leftarrow I \,\|\, N$
$V_0 \leftarrow \mathsf{HASH\text{-}DRBG\_df}(seed\_material, L)$
$C_0 \leftarrow \mathsf{HASH\text{-}DRBG\_df}(\text{0x00} \,\|\, V_0, L)$
$cnt_0 \leftarrow 1$
Return (V_0, C_0, cnt_0)

HASH-DRBG refresh
Require: $S = (V, C, cnt), I$
Ensure: $S' = (V', C', cnt')$
$seed\_material \leftarrow \text{0x01} \,\|\, V \,\|\, I$
$V' \leftarrow \mathsf{HASH\text{-}DRBG\_df}(seed\_material, L)$
$C' \leftarrow \mathsf{HASH\text{-}DRBG\_df}(\text{0x00} \,\|\, V', L)$
$cnt' \leftarrow 1$
Return (V', C', cnt')

HASH-DRBG next
Require: $S = (V, C, cnt), \beta, addin$
Ensure: $R, S' = (V', C', cnt')$
1. If $cnt > reseed\_interval$
2. Return reseed_required
3. If $addin \neq \varepsilon$
4. $w \leftarrow \mathsf{H}(\text{0x02} \,\|\, V \,\|\, addin)$
5. $V \leftarrow (V + w) \bmod 2^L$
6. $data \leftarrow V$; $temp_R \leftarrow \varepsilon$; $n \leftarrow \lceil \beta/\ell \rceil$
7. For $j = 1, \ldots, n$
8. $r \leftarrow \mathsf{H}(data)$
9. $data \leftarrow (data + 1) \bmod 2^L$
10. $temp_R \leftarrow temp_R \,\|\, r$
11. $R \leftarrow \text{left}(temp_R, \beta)$
12. $H \leftarrow \mathsf{H}(\text{0x03} \,\|\, V)$
13. $V' \leftarrow (V + H + C + cnt) \bmod 2^L$
14. $C' \leftarrow C$; $cnt' \leftarrow cnt + 1$
15. Return $R, (V', C', cnt')$

CTR-DRBG update
Require : $provided\_data, K, V$
Ensure : K, V
$temp \leftarrow \varepsilon$; $m \leftarrow \lceil (\kappa + \ell)/\ell \rceil$
For $j = 1, \ldots, m$
  $V \leftarrow (V + 1) \bmod 2^\ell$; $Z \leftarrow \mathsf{E}(K, V)$
  $temp \leftarrow temp \,\|\, Z$
$temp \leftarrow \text{left}(temp, (\kappa + \ell))$
$temp \leftarrow temp \oplus provided\_data$
$K \leftarrow \text{left}(temp, \kappa)$
$V \leftarrow \text{right}(temp, \ell)$
Return K, V

HMAC-DRBG update
Require : $provided\_data, K, V$
Ensure: K, V
$K \leftarrow \mathsf{HMAC}(K, V \,\|\, \text{0x00} \,\|\, provided\_data)$
$V \leftarrow \mathsf{HMAC}(K, V)$
If $provided\_data \neq \varepsilon$
  $K \leftarrow \mathsf{HMAC}(K, V \,\|\, \text{0x01} \,\|\, provided\_data)$
  $V \leftarrow \mathsf{HMAC}(K, V)$
Return (K, V)

HMAC-DRBG setup
Require I, N
Ensure $S_0 = (K_0, V_0, cnt_0)$
$seed\_material \leftarrow I \,\|\, N$
$K \leftarrow \text{0x00} \ldots 00$
$V \leftarrow \text{0x01} \ldots 01$
$(K_0, V_0) \leftarrow \mathsf{update}(seed\_material, K, V)$
$cnt_0 \leftarrow 1$
return (K_0, V_0, cnt_0)

HMAC-DRBG refresh
Require: $S = (K, V, cnt), I$
Ensure: $S' = (K', V', cnt')$
$seed\_material \leftarrow I$
$(K_0, V_0) \leftarrow \mathsf{update}(seed\_material, K, V)$
$cnt_0 \leftarrow 1$
Return (K_0, V_0, cnt_0)

HMAC-DRBG next
Require: $S = (K, V, cnt)$, $\beta, addin$
Ensure: $R, S' = (K', V', cnt')$
1. If $cnt > reseed\_interval$
2. Return reseed_required
3. If $addin \neq \varepsilon$
4. $(K, V) \leftarrow \mathsf{update}(addin, K, V)$
5. $temp \leftarrow \varepsilon$; $n \leftarrow \lceil \beta/\ell \rceil$
6. For $j = 1, \ldots, n$
7. $V \leftarrow \mathsf{HMAC}(K, V)$
8. $temp \leftarrow temp \,\|\, V$
9. $R \leftarrow \text{left}(temp, \beta)$
10.$(K', V') \leftarrow \mathsf{update}(addin, K, V)$
11. $cnt' \leftarrow cnt + 1$
12. Return $R, (K', V', cnt')$

CTR-DRBG next
Require: $S = (K, V, cnt), \beta, addin$
Ensure: $R, S' = (K', V', cnt')$
1. If $cnt > reseed\_interval$
2. Return reseed_required
3. If $addin \neq \varepsilon$
4. If derivation function used then
5. $addin \leftarrow \mathsf{CTR\text{-}DRBG\_df}(addin, (\kappa + \ell))$
6. Else if $len(addin) < (\kappa + \ell)$ then
7. $addin \leftarrow addin \,\|\, 0^{(\kappa + \ell - len(addin))}$
8. $(K, V) \leftarrow \mathsf{update}(addin, K, V)$
9. Else $addin \leftarrow 0^{\kappa + \ell}$
10. $temp \leftarrow \varepsilon$; $n \leftarrow \lceil \beta/\ell \rceil$
11. For $j = 1, \ldots, n$
12. $V \leftarrow (V + 1) \bmod 2^\ell$; $r \leftarrow \mathsf{E}(K, V)$
13. $temp \leftarrow temp \,\|\, r$
14. $R \leftarrow \text{left}(temp, \beta)$
15. $(K', V') \leftarrow \mathsf{update}(addin, K, V)$
16. $cnt' \leftarrow cnt + 1$
17. Return $R, (K', V', cnt')$

Fig. 1: Component algorithm for HASH-DRBG, HMAC-DRBG and CTR-DRBG.

5 Robustness in the Random Oracle Model

Security claims. As discussed in Section 1, the stronger security properties of backtracking and prediction resistance claimed in the standard have never been formally investigated. To address this, we analyze HASH-DRBG and HMAC-DRBG in the robustness framework of [16]. This models a powerful attacker who is able to compromise the state and influence the entropy source of the PRNG, and encapsulates both backtracking and prediction resistance. We first define the notions of robustness and forward security for PRNGs [16], then introduce the notion of robustness in the ROM.

Distribution sampler. We model the gathering of entropy inputs from the entropy source via a *distribution sampler* [16]. Formally, a distribution sampler $\mathcal{D} : \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^* \times \mathbb{R}^{\geq 0} \times \{0,1\}^*$ is a stateful and probabilistic algorithm which takes as input its current state $\sigma \in \{0,1\}^*$ and outputs a tuple (σ', I, γ, z), where $\sigma' \in \{0,1\}^*$ denotes the updated state of the sampler, $I \in \{0,1\}^*$ denotes the entropy sample, $\gamma \in \mathbb{R}^{\geq 0}$ is an entropy estimate for the sample, and $z \in \{0,1\}^*$ denotes a string of side information about the sample. We say that a sampler $\mathcal{D}$ is $(q_{\mathcal{D}}^+, \gamma^*)$-*legitimate* if **(1)** for all $j \in [1, q_{\mathcal{D}} + 1]$:

$$\mathrm{H}_\infty(I_j | I_1, \ldots, I_{j-1}, I_{j+1}, \ldots, I_{q_{\mathcal{D}}+1}, \gamma_1, \ldots, \gamma_{q_{\mathcal{D}}+1}, z_1, \ldots, z_{q_{\mathcal{D}}+1}) \geq \gamma_j \ ,$$

where $\sigma_0 = \varepsilon$ and $(\sigma_j, I_j, \gamma_j, z_j) \leftarrow\!\!\$\ \mathcal{D}(\sigma_{j-1})$; and **(2)** it holds that $\gamma_1 \geq \gamma^*$. Condition **(2)** extends the definition of [16] to model the sample (which recall must contain γ^* bits of entropy) with which the DRBG is initially seeded during setup. It is straightforward to see that to any sequence of *Get_entropy_input*() calls made by the DRBG, we can define an associated sampler[4].

5.1 Robustness and Forward Security in the Random Oracle Model

Our positive results about HASH-DRBG and HMAC-DRBG will be in the random oracle model (ROM). As such, the first step in our analysis is to adapt the security model of Dodis et al. [16] to the (ROM).

Robustness and forward security. Consider the game Rob shown in Figure 2. The game is parameterized by an entropy threshold γ^*. We expect security when the entropy in the system is at least this value. Mapping the NIST DRBGs into this model, we take γ^* equal to the security strength of the implementation. At the start of the game, we choose a random function $\mathsf{H} \leftarrow\!\!\$\ \mathcal{H}$ where $\mathcal{H}$ denotes the set of all functions of a given domain and range. All of the PRNG algorithms have access to H which we indicate in superscript (e.g., $\mathsf{setup}^{\mathsf{H}}$). Unlike the modelling of robustness in the ideal permutation model of Gazi et al. [20] we do *not* give the sampler $\mathcal{D}$ access to H for reasons we discuss below. To the best of our knowledge this is the first work to consider robustness in the ROM, and our security model may be useful to analyze other PRNGs beyond HASH-DRBG and HMAC-DRBG. We have additionally modified game Rob from [16] to: **(1)** accommodate our PRNG syntax (including the use of additional input, discussed below); **(2)** remove the Next oracle, which was shown in [14] to be without loss of generality; and **(3)** generate the initial state via the $\mathsf{setup}^{\mathsf{H}}$ algorithm (as opposed to initializing the PRNG with an ideal random state) similarly to [38].

The game is implicitly parameterized by the nonce distribution $\mathcal{N}$ used by $\mathcal{G}$, where we write $N \leftarrow \mathcal{N}$ to denote sampling a nonce. Since nonces may be predictable (e.g., if a sequence number is used) we assume $\mathcal{N}$ is public and that the nonce sampled by the challenger at the start of an execution of Rob is given to $\mathcal{A}$. Similarly, we assume the attacker provides the strings of additional input which may be provided in next calls. These are conservative assumptions, since any entropy in these values can only make the attacker's job harder. We assume that an attacker either always or never includes additional input in RoR queries.

We define game Fwd to be a restricted variant of game Rob, in which the attacker $\mathcal{A}$ can make no Set queries and makes a single Get query after which they may make no further queries. With

[4] The SP 800-90B standard defines the entropy estimate of sample I as simply $\mathrm{H}_\infty(I)$, as opposed to conditioning on other samples and associated data. However, the tests specified in SP 800-90B estimate entropy using multiple samples drawn from the source. As such, it seems reasonable to assume that entropy sources satisfy the conditional entropy requirement used above.

this in place, the *robustness* advantage of a legitimate distribution sampler $(q_{\mathcal{D}}^{+}, \gamma^*)$-legitimate $\mathcal{D}$ and adversary $\mathcal{A}$ is

$$\mathrm{Adv}^{\mathrm{rob}}_{\mathcal{G},\gamma^*}(\mathcal{A},\mathcal{D}) = 2 \cdot |\Pr\left[\,\mathrm{Rob}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\gamma^*} \Rightarrow 1\,\right] - \frac{1}{2}| \,,$$

and the *forward security* advantage of $(\mathcal{A},\mathcal{D})$ is defined

$$\mathrm{Adv}^{\mathrm{fwd}}_{\mathcal{G},\gamma^*}(\mathcal{A},\mathcal{D}) = 2 \cdot |\Pr\left[\,\mathrm{Fwd}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\gamma^*} \Rightarrow 1\,\right] - \frac{1}{2}| \,.$$

In both cases, we say that $\mathcal{A}$ is a $(q_{\mathsf{H}}, q_R, q_{\mathcal{D}}, q_S)$-adversary if it makes q_{H} queries to the random oracle H, q_R queries to its RoR oracle, $q_{\mathcal{D}}$ queries to its Ref oracle, and q_S queries to its Get / Set oracles. It is straightforward to see that any PRNG which is robust is also forward secure.

Variants. Games Rob and Fwd in the standard model are defined identically to the corresponding notions in the ROM, except we no longer sample a random oracle $\mathsf{H} \leftarrow\!\!\$\ \mathcal{H}$ at the start of the game and remove oracle access to H from all algorithms. In much of our analysis we make a simplifying assumption that the attacker always requests outputs of some fixed length β in RoR queries. We indicate this by adding β in subscript e.g., $\mathrm{Rob}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\gamma^*,\beta}$. This is to avoid further complicating security bounds with parameters indicating the length of the output requested in each RoR query. The analogous results for the fully general game Rob can be recovered as a straightforward extension of our proofs.

The problem of salting. It is well-known that deterministic extraction from imperfect sources is impossible in general, which is why the PRNG in game Rob is initialized with a random salt X which crucially is chosen independently of the input distribution. Unfortunately (for our analysis) none of the NIST DRBGs are specified to take a salt. Moreover, HMAC-DRBG and HASH-DRBG have no state components or inputs which can be reframed as a salt without adding substantial assumptions. Indeed, any salt derived from the entropy source will not satisfy the required independence criteria. For example, this rules out reframing the constant C which is a state component of HASH-DRBG as a salt. Likewise, the standard allows the nonce used by setup to be sampled from the source, rendering this unsuitable also. While we omit the optional personalization string (input to setup) and optional additional input (input to refresh) from our analysis for simplicity, we stress that since these are provided by the caller and are arbitrary these would not in general be suitable had they been included. For example, there is nothing to prevent the caller using past PRNG outputs, which clearly depend on the source, as additional input.

At this point we are faced with two choices. We either: **(1)** allow the sampler $\mathcal{D}$ to query the random oracle H (similarly to the notion of oracle-dependent samplers in the ideal permutation model of [20]). To construct a security proof in this case we must either modify the NIST DRBGs to accommodate a random salt or restrict our analysis to implementations for which the nonce / personalization string and additional input are sufficiently independent of the entropy source to be framed as a salt. Or: **(2)** do not allow $\mathcal{D}$ to query the random oracle. In this case, the oracle H which security analysis is with respect to is chosen randomly and independently of the entropy source, and so serves the same purpose as a random salt. We have opted to take the latter approach for a number of reasons. Firstly, we wish to analyze the NIST DRBGs as they are specified and used. As such modifying the construction or greatly restricting the number of implementations we can reason about as per **(1)** solely to facilitate the analysis seems counterproductive. Secondly, as pointed out in [39], generating a salt is challenging in practice, due to the necessary independence from the entropy source. Moreover, given the litany of tests which approved entropy sources in SP 800-90B are subjected to, it seems reasonable to assume that no source which passes these tests will be so biased as to be problematic.

5.2 Preserving and Recovering Security in the ROM

A key insight of [16] is that the complex notion of robustness can be decomposed into two simpler notions called preserving and recovering security. The former models the PRNGs ability to maintain security if the state is secret but the attacker is able to influence the entropy source. The latter models the PRNGs ability to recover from state compromise after sufficient (honestly generated)

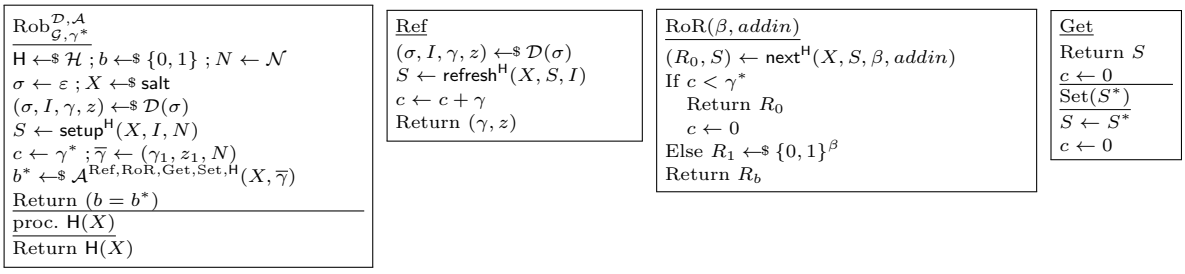

Fig. 2: Security game Rob for a PRNG $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$.

entropy has entered the system. Here we will utilize the variants of these from [38], which extended the original definitions and added a new game Init modelling initial state generation.

Consider the games Pres, Rec, and Init shown in Figure 3[5]. All games are defined with respect to a *masking function*, which is a randomized function $\mathsf{M} : \mathcal{S} \cup \{\varepsilon\} \to \mathcal{S}$ where $\mathcal{S}$ denotes the state space of the PRNG[6]. Here we have adapted the notions of [38] in the natural way to accommodate: **(1)** a random oracle; and **(2)** our PRNG syntax. We give the masking function access to the random oracle, indicated by M^{H}. We make two further modifications. Firstly in Init, we require S_0^* to be indistinguishable from $\mathsf{M}^{\mathsf{H}}(\varepsilon)$ as opposed to $\mathsf{M}^{\mathsf{H}}(S_0^*)$ as in [38]. Secondly, during the computation of the challenge in Pres and Rec, we apply the masking function to the state S_d which was *input* to $\mathsf{next}^{\mathsf{H}}$ as opposed to the state S^* *output* by $\mathsf{next}^{\mathsf{H}}$. In both cases, this is to accommodate the somewhat complicated state distribution of HASH-DRBG (see Section 6). For all $\mathrm{Gm}_y^x \in \{\mathrm{Init}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*}, \mathrm{Pres}^{\mathcal{A},}_{\mathcal{G},\mathsf{M},\beta}, \mathrm{Rec}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta}\}$ we define

$$\mathrm{Adv}^{\mathrm{gm}}_x(y) = 2 \cdot |\Pr\left[\, \mathrm{Gm}_y^x \Rightarrow 1 \,\right] - \frac{1}{2}| \,,$$

where $\mathcal{A}$ is said to be a q_{H} adversary if they make q_{H} queries to their H oracle. With this in place, the following theorem — which says that Init, Pres and Rec security collectively imply Rob security in the ROM — is an adaptation of the analogous results from [38], [20]. As a bonus, employing a slightly different line of argument with two series of hybrid arguments means our proof holds for arbitrary masking functions, lifting the restriction from [38] that masking functions possess a property called idempotence. We provide a proof in Appendix D.

Theorem 1. *Let $\mathcal{G} = (\mathsf{setup}^{\mathsf{H}}, \mathsf{refresh}^{\mathsf{H}}, \mathsf{next}^{\mathsf{H}})$ be a PRNG with input, built from a hash function H which we model as a random oracle. Suppose that each invocation of $\mathsf{refresh}^{\mathsf{H}}$ and $\mathsf{next}^{\mathsf{H}}$ makes at most q_{ref} and q_{nxt} queries to H respectively. Let $\mathsf{M}^{\mathsf{H}} : \mathcal{S} \cup \{\varepsilon\} \to \mathcal{S}$ be a masking function for which each invocation of H makes at most q_{M} H queries. Then for any $(q_{\mathsf{H}}, q_{\mathcal{D}}, q_R, q_S)$-adversary $\mathcal{A}$ and $(q_{\mathcal{D}}^+, \gamma^*)$-legitimate sampler $\mathcal{D}$ in game Rob against $\mathcal{G}$, there exists a $(q_{\mathsf{H}} + q_{\mathcal{D}} \cdot q_{ref} + q_R \cdot q_{nxt})$-adversary $\mathcal{A}_1$ and $(q_{\mathsf{H}} + q_{\mathcal{D}} \cdot q_{ref} + q_R \cdot (q_{nxt} + q_{\mathsf{M}}))$-adversaries $\mathcal{A}_2, \mathcal{A}_3$ such that*

$$\mathrm{Adv}^{\mathrm{rob}}_{\mathcal{G},\gamma^*,\beta}(\mathcal{A}, \mathcal{D}) \leq 2 \cdot \mathrm{Adv}^{\mathrm{init}}_{\mathcal{G},\mathsf{M},\gamma^*}(\mathcal{A}_1, \mathcal{D}) + 2q_R \cdot \mathrm{Adv}^{\mathrm{pres}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}_2) + 2q_R \cdot \mathrm{Adv}^{\mathrm{rec}}_{\mathcal{G},\mathsf{M},\beta,\gamma^*}(\mathcal{A}_3, \mathcal{D}) \,.$$

Tightness. Unfortunately due to a hybrid argument taken over the q_R RoR queries made by $\mathcal{A}$, Theorem 1 is not tight. This is exacerbated in the ROM, since the attacker in each of the q_R hybrid reductions must make enough H queries to simulate the whole of game Rob for $\mathcal{A}$. This hybrid argument accounts for the q_R coefficients in the bound and in the attacker query budgets. This seems inherent to the proof technique and is present in the analogous results of [16, 38, 20]. Developing a technique to obtain tighter bounds is an important open question.

[5] To avoid further complicating our analysis, the notions given here are for the variant of Rob in which $\mathcal{A}$ always requests outputs of β-bits in RoR queries. It is straightforward to extend our analysis to accommodate variable length outputs.

[6] We extend the definition of [38] to include the empty string ε, and discuss the reasons for this in Section 6. We assume that ε does not lie in the state space of the PRNG; if this is not the case then any distinguished symbol may be used instead.

$\underline{\text{Init}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*}}$
$\mathsf{H} \leftarrow\!\$\ \mathcal{H}\ ; b \leftarrow\!\$\ \{0,1\}\ ; N \leftarrow \mathcal{N}$
$\sigma_0 \leftarrow \varepsilon\ ; X \leftarrow\!\$\ \mathsf{salt}$
For $k = 1, \ldots, q_{\mathcal{D}} + 1$
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow\!\$\ \mathcal{D}(\sigma_{k-1})$
If $(b = 0)$ then $S_0^* \leftarrow \mathsf{setup}^{\mathsf{H}}(X, I_1, N)$
Else $S_0^* \leftarrow\!\$\ \mathsf{M}^{\mathsf{H}}(\varepsilon)$
$b^* \leftarrow\!\$\ \mathcal{A}^{\mathsf{H}}(X, S_0^*, (I_i)_{i=2}^{q_{\mathcal{D}}+1}, (\gamma_i, z_i)_{i=1}^{q_{\mathcal{D}}+1}, N)$
Return $(b = b^*)$

$\underline{\text{Pres}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta}}$
$\mathsf{H} \leftarrow\!\$\ \mathcal{H}\ ; b \leftarrow\!\$\ \{0,1\}$
$X \leftarrow\!\$\ \mathsf{salt}$
$(S_0', I_1, \ldots, I_d, addin) \leftarrow\!\$\ \mathcal{A}^{\mathsf{H}}(X)$
$S_0 \leftarrow\!\$\ \mathsf{M}^{\mathsf{H}}(S_0')$
For $i = 1, \ldots, d$
  $S_i \leftarrow \mathsf{refresh}^{\mathsf{H}}(X, I_i, S_{i-1})$
If $(b = 0)$ then $(R^*, S^*) \leftarrow \mathsf{next}^{\mathsf{H}}(X, S_d, \beta, addin)$
Else $R^* \leftarrow \{0,1\}^{\beta}\ ; S^* \leftarrow \mathsf{M}^{\mathsf{H}}(S_d)$
$b^* \leftarrow\!\$\ \mathcal{A}^{\mathsf{H}}(X, R^*, S^*)$
Return $(b = b^*)$

$\underline{\text{Rec}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta}}$
$\mathsf{H} \leftarrow\!\$\ \mathcal{H}\ ; b \leftarrow\!\$\ \{0,1\}$
$\sigma \leftarrow \varepsilon\ ; X \leftarrow\!\$\ \mathsf{salt}\ ; \mu \leftarrow 1$
For $k = 1, \ldots, q_{\mathcal{D}} + 1$
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow\!\$\ \mathcal{D}(\sigma_{k-1})$
$(S_0, d, addin) \leftarrow\!\$\ \mathcal{A}^{\mathsf{H},\text{Sam}}(X, (I_1, (\gamma_k, z_k))_{i=1}^{q_{\mathcal{D}}+1})$
If $\mu + d > (q_{\mathcal{D}} + 1)$ or $\sum_{i=\mu+1}^{\mu+d} \gamma_i < \gamma^*$
  Return $\perp$
For $i = 1, \ldots, d$
  $S_i \leftarrow \mathsf{refresh}^{\mathsf{H}}(X, I_{\mu+i}, S_{i-1})$
If $(b = 0)$ then $(R^*, S^*) \leftarrow\!\$\ \mathsf{next}^{\mathsf{H}}(X, S_d, \beta, addin)$
$R^* \leftarrow \{0,1\}^{\beta}\ ; S^* \leftarrow\!\$\ \mathsf{M}^{\mathsf{H}}(S_d)$
$b^* \leftarrow\!\$\ \mathcal{A}(X, R^*, S^*, (I_k)_{k>\mu+d})$
Return $(b = b^*)$

$\underline{\text{Sam}()}$
$\mu = \mu + 1$
Return I_μ

$\underline{\text{proc. } \mathsf{H}(X)}$
Return $\mathsf{H}(X)$

Fig. 3: Security games Init, Pres and Rec for a PRNG $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ and $\mathsf{M} : \mathcal{S} \cup \{\varepsilon\} \to \mathcal{S}$.

$\underline{\mathsf{M}^{\mathsf{H}}(S)}$
If $S = \varepsilon$
  $V' \leftarrow\!\$\ \{0,1\}^L$
  $C' \leftarrow \mathsf{HASH\text{-}DRBG\_df}^{\mathsf{H}}(0\text{x}00||V, L)$
  $cnt' \leftarrow 1$
Else $(V, C, cnt) \leftarrow S$
  $H \leftarrow\!\$\ \{0,1\}^{\ell}$
  $V' \leftarrow (V + C + cnt + H) \bmod 2^L$
  $C' \leftarrow C\ ; cnt' \leftarrow cnt + 1$
$S' \leftarrow (V', C', cnt')$
Return S'

$\underline{\mathsf{M}^{\mathsf{HMAC}}(S)}$
$K', V' \leftarrow\!\$\ \{0,1\}^{\ell}$
If $S = \varepsilon$
  $cnt' \leftarrow 1$
Else $(K, V, cnt) \leftarrow S$
  $cnt' \leftarrow cnt + 1$
$S' \leftarrow (K', V', cnt')$
Return S'

Fig. 4: Masking function for proofs of Theorem 2 and Theorem 4.

6 Analysis of HASH-DRBG

We now present our analysis of the robustness of HASH-DRBG, in which the underlying hash function $\mathsf{H} : \{0,1\}^{\leq\omega} \to \{0,1\}^{\ell}$. Our proof is with respect to the masking function M^{H} shown in the left hand side of Figure 4. To avoid further complicating security bounds, we assume that HASH-DRBG is never called with additional input and that β-bits are requested in each RoR query. It is straightforward to generalize the result to include variable output lengths, and we expect the proof for HASH-DRBG called with additional input to be very similar. Since HASH-DRBG is not specified to take a salt, we omit this parameter in the subsequent exposition.

Challenges. Certain design aspects of HASH-DRBG significantly complicate the proof, and necessitated adaptations in our security modeling (Section 5).

- Notice that the distributions of states returned by $\mathsf{setup}^{\mathsf{H}}$ and $\mathsf{refresh}^{\mathsf{H}}$ are quite different from the distribution of states S' where $(R, S') \leftarrow \mathsf{next}^{\mathsf{H}}(S, \beta)$. To accommodate this, we extend the domain of M to include the empty string ε to indicate that an idealized state of the first form should be returned. Juggling these different state distributions complicates proofs, in particular introducing multiple cases into the proof of Pres security.
- Consider the distribution of $S' = \mathsf{M}^{\mathsf{H}}(S)$ for $S \in \mathcal{S}$, which serves to idealize the distribution of the state $S' = (V', C', cnt')$ as updated following an output generation request. It would be convenient to argue that V' is uniformly distributed over $\{0,1\}^L$. However, since V' is chosen uniformly from the window $[V + C + cnt, V + C + cnt + (2^{\ell} - 1)]$ where $S = (V, C, cnt)$ and $L > \ell$ (see below), this is clearly not the case. Since the range in which the updated state S' may lie is dependant on the previous state S in this way, we adapted games Pres and Rec so that it is S which is masked instead of S'. For the same reason, bounding the probability that $\mathcal{A}$ guesses the previous state S when given S' requires some care.

– More minor issues, such as: **(1)** not properly separating the domain of queries made by $\mathsf{setup}^{\mathsf{H}}$ to produce the counter V from those made to produce the constant C; and **(2)** the way in which L is not a multiple of ℓ for the approved hash functions; make certain steps in the analysis more fiddly than they might have been.

Parameter settings. We provide a general treatment here to which any parameter setting may be slotted in, subject to two restrictions which are utilized in the proof. Namely, we assume that $L > \ell + 1$ and $n < 2^L$, where $n = \lceil \beta/\ell \rceil$ denotes the number of output blocks produced by $\mathsf{next}^{\mathsf{H}}$ to satisfy a request for β-bits. We additionally require that $L < 2^{32}$ and $m < 2^8$ where $|V| = |C| = L$ and $m = \lceil L/\ell \rceil$ is the number of blocks hashed by $\mathsf{setup}^{\mathsf{H}}$ / $\mathsf{refresh}^{\mathsf{H}}$ to produce a new counter. This is because these values have to be encoded as a 32-bit and 8-bit string respectively by HASH-DRBG_df. All hash functions approved in the standard fall well within these parameters. (Indeed, for all of these $L > 2\ell$, $n < 3277 \ll 2^L$, and $m \leq 3$.)

Proof of robustness. With this in place, we present the following theorem bounding the robustness of HASH-DRBG. The proof follows from a number of lemmas which we discuss below, combined with Theorem 1. (When calculating the query budgets for Theorem 1, it is readily verified that for HASH-DRBG $q_{nxt} = n + 1$, $q_{ref} = 2m$, and $q_{\mathsf{M}} = m$.)

Theorem 2. *Let $\mathcal{G}$ be* HASH-DRBG *built from a hash function $\mathsf{H} : \{0,1\}^{\leq \omega} \to \{0,1\}^{\ell}$ which we model as a random oracle. Let L denote the state length of the instantiation where $L > \ell + 1$, let $m = \lceil L/\ell \rceil$, and suppose that* HASH-DRBG *is not called with additional input. Let M^{H} denote the masking function shown in the left-hand panel of Figure 4. Then for any $(q_{\mathsf{H}}, q_{\mathcal{D}}, q_R, q_S)$-attacker $\mathcal{A}$ in the robustness game against $\mathcal{G}$, and any $(q_{\mathcal{D}}^{+}, \gamma^*)$-legitimate sampler $\mathcal{D}$, it holds that*

$$\mathrm{Adv}^{\mathrm{rob}}_{\mathcal{G},\mathsf{M},\beta,\gamma^*}(\mathcal{A},\mathcal{D}) \leq \frac{q_R \cdot \overline{q}_{\mathsf{H}} + 2q'_{\mathsf{H}}}{2^{\gamma^*-2}} + \frac{q_R \cdot \overline{q}_{\mathsf{H}} \cdot (2n+1)}{2^{\ell-2}} + \frac{q_R \cdot ((d-1)(2\overline{q}_{\mathsf{H}} + d) + \overline{q}_{\mathsf{H}}^2) + 2}{2^{L-2}} \ .$$

Here $n = \lceil \beta/\ell \rceil$ and d denotes the maximum number of consecutive Ref queries. Moreover, $q'_{\mathsf{H}} = (q_{\mathsf{H}} + 2m \cdot q_{\mathcal{D}} + (n+1) \cdot q_R)$ and $\overline{q}_{\mathsf{H}} = q'_{\mathsf{H}} + m \cdot q_R$.

Init security. We begin by bounding the Init security of HASH-DRBG. The $q_{\mathsf{H}} \cdot 2^{-\gamma^*}$ term arises following the standard argument that the initial state variable V_0 will be indistinguishable from a truly random bit string unless the attacker can guess the entropy sample I_1 which was hashed to produce it. The additional 2^{-L} term arises since the queries made to compute the counter V_0 are not fully domain separated from those made to compute the constant C_0. Indeed, if it so happens that $I_1 || N = \mathtt{0x00} || V_0$ where I_1 and N denote the entropy input and nonce (an event which — while very unlikely — is not precluded by the parameter settings in the standard), then the derived values of V_0 and C_0 will be equal, allowing the attacker to distinguish the real state from $\mathsf{M}^{\mathsf{H}}(\varepsilon)$ with high probability. A small tweak to the design of setup (e.g., prepending $\mathtt{0x01}$ to $I||N$ before hashing) would have avoided this. The full proof is given in Appendix E.

Lemma 1. *Let $\mathcal{G}$ =* HASH-DRBG *and masking function M^{H} be as specified in Theorem 2. Then for any adversary $\mathcal{A}$ in game Init against $\mathcal{G}$ making q_{H} queries to the random oracle H, and any $(q_{\mathcal{D}}^{+}, \gamma^*)$-legitimate sampler $\mathcal{D}$, it holds that*

$$\mathsf{Adv}^{\mathrm{init}}_{\mathcal{G},\mathsf{M},\gamma^*}(\mathcal{A},\mathcal{D}) \leq q_{\mathsf{H}} \cdot 2^{-\gamma^*} + 2^{-L} \ .$$

Pres security. At the start of game Pres, the attacker outputs $(S'_0, I_1, \ldots, I_d)$. The game sets $(V_0, C_0, cnt_0) \leftarrow\!\!\$\ \mathsf{M}^{\mathsf{H}}(S'_0)$, and iteratively computes S_d via $S_i = \mathsf{refresh}^{\mathsf{H}}(S_{i-1}, I_i)$ for $i \in [1, d]$. The proof begins by arguing that unless the attacker can guess the counter V_0 or any of the counters $V_1, \ldots, V_{d-1}$ passed through during reseeding, then (barring certain accidental collisions) the updated state $S_d = (V_d, C_d, cnt_d)$ is indistinguishable from a masked state. The proof then shows that, unless the attacker can guess V_d, the resulting output / state pair are indistinguishable from their idealized counterparts. We must consider a number of cases depending on whether the tuple $(S'_0, I_1, \ldots, I_d)$ output by $\mathcal{A}$ is such that: **(1)** $S'_0 \in \mathcal{S}$ or $S'_0 = \varepsilon$; and **(2)** $d \geq 1$ or $d = 0$; since these induce different distributions on S_0 and S_d respectively.

Lemma 2. *Let $\mathcal{G} = \mathsf{HASH\text{-}DRBG}$ and masking function M^{H} be as specified in Theorem 2. Then for any adversary $\mathcal{A}$ in game Pres against $\mathcal{G}$ making q_{H} queries to the random oracle H, it holds that*

$$\mathrm{Adv}^{\mathrm{pres}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}) \leq \frac{q_{\mathsf{H}} \cdot (n+1)}{2^{\ell-1}} + \frac{(d-1)(2q_{\mathsf{H}}+d)}{2^{L}} ,$$

where $n = \lceil \beta/\ell \rceil$ and $\mathcal{A}$ outputs d entropy samples at the start of the challenge.

Rec security. The first step in the proof of Rec security argues that iteratively reseeding an adversarially chosen state S_0 with d entropy samples which collectively have entropy γ^* yields a state $S_d = (V_d, C_d, cnt_d)$ which is indistinguishable from $\mathsf{M}^{\mathsf{H}}(\varepsilon)$. This represents the main technical challenge in the proof, and uses Patarin's H-coefficient technique, which we recall in Appendix A.

Our proof is based on the analogous result for sponge-based PRNGs in the ideal permutation model (IPM) of Gazi et al. [20], essentially making the same step-by-step argument. However, making the necessary adaptations to analyze HASH-DRBG is still non-trivial. As well as working in the ROM as opposed to the IPM, we must adapt the proof to take into account the constant C which is a state component of HASH-DRBG, as well as the more involved reseeding process, which concatenates and truncates the responses to multiple H queries to derive each updated counter V'.

At a high level, we say that an execution of the game is *bad* if $\mathcal{A}$ makes sufficient H queries to compute V_d himself. Any such set of queries requires $\mathcal{A}$ to guess the entropy samples used for reseeding, and so contributes the $q_{\mathsf{H}} \cdot 2^{-\gamma^*}$ term. The proof then argues that if a transcript is *not* bad then, barring accidental collisions (accounting for the remaining terms in the bound), the final updated state component V_d will be uniformly distributed over $\{0,1\}^L$, and so in turn, S_d is equivalent to $\mathsf{M}^{\mathsf{H}}(\varepsilon)$. With this in place, an analogous argument to that made in the proof of Pres security, implies that an output / state pair produced by applying $\mathsf{next}^{\mathsf{H}}$ to this masked state, are indistinguishable from their idealized counterparts.

Lemma 3. *Let $\mathcal{G} = \mathsf{HASH\text{-}DRBG}$ and masking function M^{H} be as specified in Theorem 2. Then for any adversary $\mathcal{A}$ in game* Rec *against $\mathcal{G}$ making q_{H} queries to the random oracle H, and any $(q^+_{\mathcal{D}}, \gamma^*)$-legitimate sampler $\mathcal{D}$, it holds that*

$$\mathrm{Adv}^{\mathrm{rec}}_{\mathcal{G},\mathsf{M},\beta,\gamma^*}(\mathcal{A},\mathcal{D}) \leq \frac{q_{\mathsf{H}}}{2^{\gamma^*-1}} + \frac{q_{\mathsf{H}} \cdot n}{2^{(\ell-1)}} + \frac{(d-1)\cdot(2q_{\mathsf{H}}+d) + 2q_{\mathsf{H}}^2}{2^{L}} .$$

Here $n = \lceil \beta/\ell \rceil$, and d denotes the index output by $\mathcal{A}$.

7 Analysis of HMAC-DRBG

We now present our analysis of HMAC-DRBG. We give both positive and negative results, showing that the security guarantees of HMAC-DRBG differ depending on whether the DRBG is called with additional input.

7.1 Negative Result: HMAC-DRBG Without Additional Input is not Forward Secure

We present an attack which breaks the forward security of HMAC-DRBG if additional input is not always included in next calls. This contradicts the claim in the standard that the NIST DRBGs are backtracking resistant. Since Rob security implies Fwd security, this rules out a proof of robustness in this case also.

The attack. Consider the update call which is performed after output block generation in the next algorithm of HMAC-DRBG (Section 3). Notice that if $addin = \varepsilon$ then the final two lines of update are not executed. As such, one may verify that the updated state $S^* = (K^*, V^*, cnt^*)$ is of the form $V^* = \mathsf{HMAC}(K^*, r^*)$ where r^* is the final output block produced in the call[7]. An attacker $\mathcal{A}$ in game Fwd who makes a RoR query to request ℓ-bits of output followed immediately by a Get query to learn S^* can easily test this relation. If it does not hold, they know the challenge output is truly random.

[7] This observation is implicit in the proof of pseudorandomness by Hirose [22]; however, the connection to forward security is not made in this work.

To concretely bound $\mathcal{A}$'s advantage we define game Fwd$^{\$}$, which is identical to game Fwd against HMAC-DRBG in the standard model except the PRNG is initialized with an 'ideally distributed' state $S_0 = (K_0, V_0, cnt_0)$ where $K_0, V_0 \leftarrow\!\!\$\ \{0,1\}^\ell$ and $cnt_0 \leftarrow 1$, as opposed to deriving S_0 from the entropy source via setup. We note that the attacker's job can only be harder in Fwd$^{\$}$, since they cannot exploit any flaws or imperfections in the setup procedure. The proof of the following theorem is given in Appendix F.

Theorem 3. *Consider an implementation $\mathcal{G}$ of* HMAC-DRBG *built from the function* HMAC $: \{0,1\}^\ell \times \{0,1\}^{\leq\omega} \to \{0,1\}^\ell$. *Suppose that additional input is not always included in* next *calls to $\mathcal{G}$. Then there exist efficient adversaries $\mathcal{A}, \mathcal{B}$ such that for any sampler $\mathcal{D}$ it holds that*

$$\mathrm{Adv}^{\text{fwd-\$}}_{\mathcal{G},\gamma^*}(\mathcal{A},\mathcal{D}) \geq 1 - 2 \cdot \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{HMAC}}(\mathcal{B}, 2) - 2^{-(\ell-1)} \ .$$

Moreover, $\mathcal{A}$ makes one RoR query and one Get query. $\mathcal{B}$ runs in the same time as $\mathcal{A}$, and makes two queries to their real-or-random function oracle.

7.2 Positive Result: Robustness of HMAC-DRBG with Additional Input in the ROM

We prove that HMAC-DRBG is robust when additional input is used, with respect to a restricted (but realistic) class of samplers. We model the function HMAC $: \{0,1\}^\ell \times \{0,1\}^{\leq\omega} \to \{0,1\}^\ell$ as a keyed random oracle, whereby each fresh query of the form $(K, X) \in \{0,1\}^\ell \times \{0,1\}^{\leq\omega}$ is answered with an independent random ℓ-bit string.

Rationale. While a standard model proof of Pres security is possible via a reduction to the PRF-security of HMAC, how to achieve the same for Init and Rec is far from clear. These results require showing that HMAC is a good (statistical or computational) randomness extractor. In games Init and Rec, the key for HMAC is either chosen by or known to the attacker, and so we cannot appeal to the PRF-security of HMAC. Entropy samples are non-uniform, so a dual-PRF assumption does not suffice either. As such, some idealized assumption on HMAC or the underlying hash / compression function seems inherent.

The extraction properties of HMAC (under various assumptions) were studied in [15], which considers a single-use version of extraction which is weaker than what is required here. These results typically model the compression function underlying HMAC as a random function. This is a reasonable heuristic when the key for HMAC is suitable for use as a salt. However for HMAC with an adversarially chosen key (as is the case in game Rec) the heuristic seems more of a stretch. Moreover, the results of [15] which avoid the ROM require high entropy inputs containing e.g., 2ℓ-bits of min-entropy, much greater than the ℓ-bits mandated by the standard, and so are not generally applicable to real-world implementations of HMAC-DRBG.

By opting to model HMAC as a keyed RO, we can analyze HMAC with respect to the entropy levels of inputs specified in the standard (and at levels which are practical for real-world applications). This is a fairly standard assumption, having been made in various other works [27, 36, 4] in which HMAC is used with a known key, or applied to entropy samples with insufficient entropy for extraction. In [17], HMAC was proven to be indifferentiable from a random oracle for all commonly deployed parameter settings (although since robustness is a multi-stage game, the indifferentiability result cannot be applied generically here [35]).

Discussion. A standard model proof for HMAC-DRBG would certainly be a stronger and more satisfying result. However as discussed above, idealizing HMAC or the underlying hash / compression function seems inherent; a result under weaker idealized assumptions is an important open problem. Despite this, we feel our analysis is a significant forward step from existing works. Ours is the first analysis of the full specification of HMAC-DRBG; prior works omit reseeding and initialization, assuming HMAC-DRBG is initialized with a state for which $K, V \leftarrow\!\!\$\ \{0,1\}^\ell$. In reality these are constructed from the entropy source, so this is far removed from HMAC-DRBG in a real system. Our work is also the first to consider security properties stronger than the pseudorandomness of output. We hope our result is a valuable first step to progress the understanding of these widely deployed (yet little analyzed) algorithms, and provides a useful starting point for further work to extend.

Sampler. We prove robustness with respect to the class of samplers $\{\mathcal{D}\}_{\gamma^*}$ defined to be the set of $(q^+_{\mathcal{D}}, \gamma^*)$-legitimate samplers for which $\gamma_i \geq \gamma^*$ for $i \in [1, q_{\mathcal{D}} + 1]$. (In words, each sample I contains

γ^*-bits of entropy). This is a simplifying assumption. However, we stress that this is the entropy level per sample required by the standard. As such, this is precisely the restriction imposed on allowed entropy sources. It seems likely that an H-coefficient proof of Rec security similar to that of Lemma 3 would yield a fully general result.

Proof of robustness. With this in place, we present the following theorem bounding the robustness of HMAC-DRBG. The proof follows from a number of lemmas which we discuss below, combined with Theorem 1 (for which it is straightforward to verify that for HMAC-DRBG, $q_{ref} = 4$ and $q_{nxt} = n + 8$ where $n = \lceil \beta/\ell \rceil$). Our proof is with respect to the masking function $\mathsf{M}^{\mathsf{HMAC}}$ shown in Figure 4. We note that, unlike for the HASH-DRBG case, $\mathsf{M}^{\mathsf{HMAC}}$ does not make any calls to HMAC ($q_{\mathsf{M}} = 0$).

Theorem 4. *Let $\mathcal{G}$ be* HMAC-DRBG *built from the function* HMAC $: \{0,1\}^\ell \times \{0,1\}^{\leq \omega} \to \{0,1\}^\ell$ *which we model as a keyed random oracle. Let $\mathsf{M}^{\mathsf{HMAC}}$ be the masking function shown in the righthand panel of Figure 4. Then for any $(q_{\mathsf{H}}, q_{\mathcal{D}}, q_R, q_S)$-attacker $\mathcal{A}$ in the robustness game against* HMAC-DRBG *who always outputs addin $\neq \varepsilon$, and any $(q^+_{\mathcal{D}}, \gamma^*)$-legitimate sampler $\mathcal{D} \in \{\mathcal{D}\}_{\gamma^*}$, it holds that*

$$\begin{aligned}\mathrm{Adv}^{\mathrm{rob}}_{\mathcal{G},\mathsf{M},\beta,\gamma^*}(\mathcal{A}) \leq\ & q_R \cdot (\bar{q}_{\mathsf{H}} \cdot \epsilon_1 + \epsilon_2) \cdot 2^{-(2\ell-1)} + \bar{q}_{\mathsf{H}} \cdot 2^{-(\ell-2)} \\ & + q_R \cdot (\bar{q}_{\mathsf{H}} \cdot (n+3) + \epsilon_3) \cdot 2^{-(\ell-2)} \\ & + (\bar{q}_{\mathsf{H}} \cdot (2q_R + (1 + 2^{-2\ell})) \cdot 2^{-(\gamma^*-1)} + 2^{-(2\ell-1)}\ .\end{aligned}$$

Here $\epsilon_1 = 12d + 10 + (4d-2) \cdot 2^{-\gamma^*}$, $\epsilon_2 = (d \cdot (10d + 4n + 18 + (d-1) \cdot 2^{-(\gamma^*-1)}) + 6n + 16)$, and $\epsilon_3 = n(n+1)$. Moreover, $n = \lceil \beta/\ell \rceil$, d denotes the maximum number of consecutive Ref queries, and $\bar{q}_{\mathsf{H}} = (q_{\mathsf{H}} + 4 \cdot q_{\mathcal{D}} + (n+8) \cdot q_R)$.

Discussion. As a concrete example, if HMAC-DRBG is instantiated with HMAC-SHA-512 then $\ell = 512$. In this case, the bound is dominated by the $\mathcal{O}(\bar{q}_{\mathsf{H}} \cdot q_R) \cdot 2^{-(\gamma^*-1)}$ term where γ^* denotes the strength of the instantiation and q_R and q_{H} correspond respectively to the number of RoR and HMAC queries made in game Rob. Supposing $q_{\mathcal{D}} \leq q_R$ (e.g., there are fewer Ref than RoR calls) and n is small, then $\bar{q}_{\mathsf{H}} \cdot q_R \leq c \cdot q_R \cdot (q_{\mathsf{H}} + q_R)$ for some small constant c. As such, if HMAC-DRBG is instantiated at strength $\gamma^* = 256$, it achieves good security margins up to fairly large q_{H}, q_R. Instantiated at lower strengths the margins are less good; however, this is likely an artefact of the proof technique rather than indicating an attack.

Init security. The proof of Init security argues that unless the attacker $\mathcal{A}$ guesses the input I_1 with which HMAC-DRBG is seeded, or an intermediate key / counter computed during setup, then — barring an accidental collision in the inputs to the second and fourth HMAC queries made by setup, contributing $2^{-2\ell}$ to the bound — the resulting state is identically distributed to $\mathsf{M}^{\mathsf{HMAC}}(\varepsilon)$. A union bound over these guessing and collision probabilities then yields the lemma.

Lemma 4. *Let $\mathcal{G}$ =* HMAC-DRBG *and masking function $\mathsf{M}^{\mathsf{HMAC}}$ be as specified in Theorem 4. Then for any adversary $\mathcal{A}$ in game* Init *against* HMAC-DRBG *making q_{H} queries to the random oracle* HMAC*, and any $(q^+_{\mathcal{D}}, \gamma^*)$-legitimate sampler $\mathcal{D} \in \{\mathcal{D}\}_{\gamma^*}$, it holds that*

$$\mathrm{Adv}^{\mathrm{init}}_{\mathcal{G},\mathsf{M},\gamma^*}(\mathcal{A}, \mathcal{D}) \leq q_{\mathsf{H}} \cdot ((1 + 2^{-2\ell}) \cdot 2^{-\gamma^*} + 2^{-(\ell-1)}) + 2^{-2\ell}\ .$$

Pres and Rec security. At a high level, the proofs of Pres and Rec security proceed by bounding: **(1)** the probability that $\mathcal{A}$ guesses one of the points queried to the random oracle HMAC during the challenge computation; and **(2)** the probability of a collision amongst these points. The proof argues that if neither of these events occur, then the challenge output / state are identically distributed to their idealized counterparts. However, this process is surprisingly delicate. Firstly, the domains of queries are not fully separated, so multiple collisions must be dealt with. Secondly, the guessing and collision probabilities of points from the same domain differ throughout the game. For example, queries of the form (K, V) are made during output generation and in state updates. In the former case, the attacker knows the 'secret' counter since this doubles as an output block, whereas in the latter this is unknown. This rules out a modular treatment, and complicates the bound. This is another example of where a small modification to separate queries would simplify analysis.

Lemma 5. *Let* $\mathcal{G} = \mathsf{HMAC\text{-}DRBG}$ *and masking function* $\mathsf{M}^{\mathsf{HMAC}}$ *be as specified in Theorem 4. Then for any adversary* $\mathcal{A}$ *in game* Pres *against* HMAC-DRBG *who makes* q_{H} *queries to the random oracle* HMAC *and always outputs addin* $\neq \varepsilon$*, it holds that*

$$\mathrm{Adv}^{\mathrm{pres}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}) \leq (q_{\mathsf{H}} \cdot (8d+6) + \epsilon) \cdot 2^{-2\ell} + (q_{\mathsf{H}} \cdot (n+2) + n(n+1)) \cdot 2^{-\ell} \ ,$$

where $\epsilon = d \cdot (6d + 2n + 8) + 3n + 8$*. Here* $n = \lceil \beta/\ell \rceil$*, and* $\mathcal{A}$ *outputs* $(S'_0, I_1, \ldots, I_d, addin)$ *at the start of the challenge.*

Lemma 6. *Let* $\mathcal{G} = \mathsf{HMAC\text{-}DRBG}$ *and masking function* $\mathsf{M}^{\mathsf{HMAC}}$ *be as specified in Theorem 4. Then for any adversary* $\mathcal{A}$ *in game* Rec *against* HMAC-DRBG *making who makes* q_{H} *queries to the random oracle* HMAC *and always outputs addin* $\neq \varepsilon$*, and any* $(q^+_{\mathcal{D}}, \gamma^*)$*-legitimate sampler* $\mathcal{D} \in \{\mathcal{D}\}_{\gamma^*}$*, it holds that*

$$\begin{aligned}\mathrm{Adv}^{\mathrm{rec}}_{\mathcal{G},\mathsf{M},\beta,\gamma^*}(\mathcal{A},\mathcal{D}) \leq (q_{\mathsf{H}} \cdot (4d + 4 + (4d-2) \cdot 2^{-\gamma^*}) + \epsilon') \cdot 2^{-2\ell} \\ + (q_{\mathsf{H}} \cdot (n+4) + n(n+1)) \cdot 2^{-\ell} + q_{\mathsf{H}} \cdot 2^{-(\gamma^*-1)} \ ,\end{aligned}$$

where $\epsilon' = (d \cdot (4d + 2n + 10 + (d-1) \cdot 2^{-(\gamma^*-1)}) + 3n + 8)$*. Here* $n = \lceil \beta/\ell \rceil$*, and* d *denotes the index output by* $\mathcal{A}$*.*

8 Overlooked Attack Vectors

The positive results of Sections 6 and 7 are reassuring. However, the flexibility in the standard to produce variable length and *large* outputs (of up to 2^{19} bits) in each next call means that two implementations of the same DRBG may be very different depending on how such limits are set. While this is reflected in the security bounds of the previous sections (in terms of the parameter n denoting the number of output blocks computed per request), we argue that the standard security definitions of forward security and robustness may overlook attack vectors against the (fairly non-standard) NIST DRBGs. The points made in this section do not contradict the results of the previous sections; rather we argue that in certain (realistic) scenarios — namely when the DRBG is used to produce many output blocks per next call — it is worth taking a closer look at which points during output generation a state may be compromised.

Iterative next **algorithms.** The next algorithm of each of the NIST DRBGs has the same high-level structure (modulo slight variations which we highlight below, and which again exemplify how small design features complicate a modular treatment of these DRBGs). On input $(S, \beta, addin)$, the reseed counter cnt is first checked to ensure it does not exceed the reseed interval. Next, any additional input provided in the call is incorporated into the state, and in the case of HASH-DRBG one of the state variables is copied into an additional variable in preparation for output generation (i.e., setting $data = V$ on line 6 of the left-hand column of Figure 1). Output blocks are then produced by iteratively applying a function to the state variables (or in the case of HASH-DRBG, the copy of the state variable). Once sufficiently many blocks have been produced to satisfy the request, these blocks are concatenated and truncated to β-bits to form the returned output R, and a final state update is performed to produce the updated state S'.

$\mathsf{next}(X, S, \beta, addin)$
If $cnt > reseed\_interval$
  Return reseed_required
$(S^0, data^0) \leftarrow \mathsf{init}(X, S, \beta, addin)$
If $addin \leftarrow \varepsilon$ then $addin \leftarrow 0^n$
$temp_R \leftarrow \varepsilon$; $n \leftarrow \lceil \beta/\ell \rceil$
For $i = 1, \ldots, n$
  $(r^i, S^i, data^i) \leftarrow \mathsf{gen}(X, S^{i-1}, data^{i-1})$
  $temp_R \leftarrow temp_R \,\|\, r^i$
$R \leftarrow \mathrm{left}(temp_R, \beta)$
$S' \leftarrow \mathsf{final}(X, S^n, \beta, addin)$
Return (R, S')

Fig. 5: Iterative next algorithm for a DRBG with associated decomposition $\mathcal{C} = (\mathsf{init}, \mathsf{gen}, \mathsf{final})$. Boxed text included for CTR-DRBG only.

We would like to track the evolution of each state variable during a next call relative to the production of different output blocks, in order to reason precisely about the points at which these state components may be compromised. As such, it shall be useful to formalize this structure. To this end, we say that a DRBG has an *iterative* next *algorithm* if next may be decomposed into a tuple of subroutines $\mathcal{C} = (\mathsf{init}, \mathsf{gen}, \mathsf{final})$. Here $\mathsf{init} : \mathsf{salt} \times \mathcal{S} \times \mathbb{N}^{\leq \alpha_{out}} \times \{0,1\}^{\leq \alpha_{add}} \rightarrow$

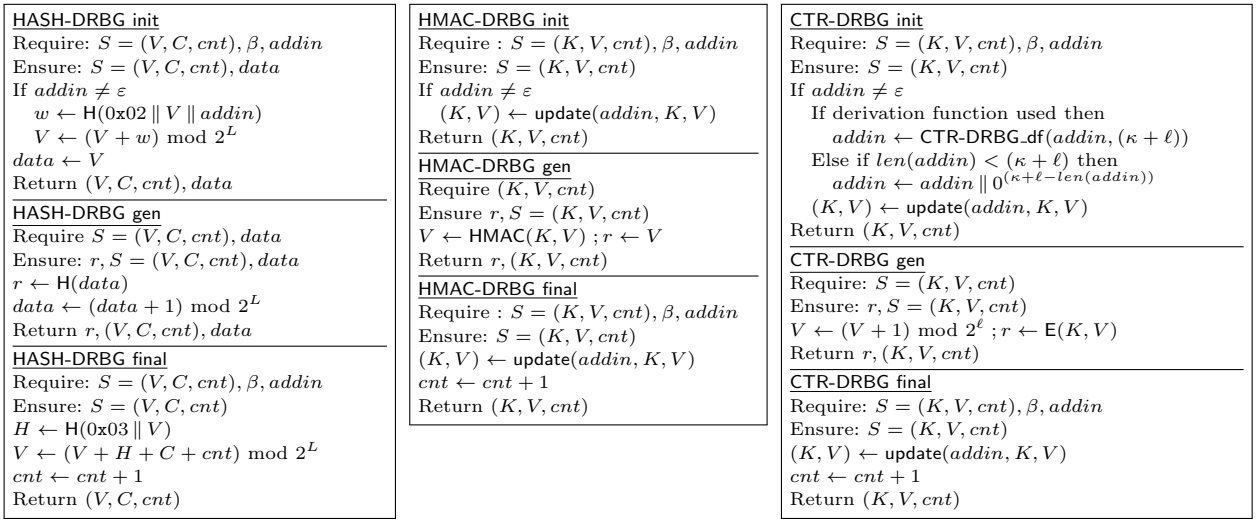

Fig. 6: Algorithms $\mathcal{C} = (\mathsf{init}, \mathsf{gen}, \mathsf{final})$ for HASH-DRBG, HMAC-DRBG and CTR-DRBG.

$\mathcal{S} \times \{0,1\}^*$ updates the state with additional input prior to output generation, and optionally sets a variable $data \in \{0,1\}^*$ to store any additional state information necessary for output generation. Algorithm $\mathsf{gen} : \mathsf{salt} \times \mathcal{S} \times \{0,1\}^* \to \{0,1\}^\ell \times \mathcal{S} \times \{0,1\}^*$ maps a state S and optional string $data$ to an output block $r \in \{0,1\}^\ell$, an updated state S', and string $data' \in \{0,1\}^*$. Finally $\mathsf{final} : \mathsf{salt} \times \mathcal{S} \times \mathbb{N}^{\leq \alpha_{out}} \times \{0,1\}^{\leq \alpha_{add}} \to \mathcal{S}$ is used to update the state post output generation. The next algorithm is constructed from these component parts as shown in Figure 5. (We note that for CTR-DRBG, we include an additional step (shown in boxed text) after the application of init in the case that $addin = \varepsilon$, to set $addin = 0^n$. This is in preparation for $addin$ later forming an input to the CTR-DRBG update function.) Looking ahead, when we discuss state leakage 'within a next call' we mean full or partial leakage of one of the pairs $(S^0, data^0), \ldots, (S^n, data^n)$ passed through during the iterative output generation process.

The component algorithms for each of the NIST DRBGs are shown in Figure 6; it is readily verified that substituting these into the framework of Figure 5 is equivalent to the next algorithm shown in Figure 1. (For CTR-DRBG and HMAC-DRBG, $data$ is not set during output generation (e.g., $data = \varepsilon$), and so we omit it from the discussion of these DRBGs. Similarly since none of the NIST DRBGs are specified to take a salt, we omit this parameter.) A diagrammatic depiction of output generation for the DRBGs is shown in Figures 7 - 9.

Variable length outputs. Within this iterative structure, the gen subroutine acts as an internal deterministic PRG, called multiple times within a single next call to produce output blocks. However, as we shall see, the state updates performed by the gen subroutine do *not* provide forward security after each block. (This is similar to an observation by Bernstein [7] which appeared concurrently to the production of the first draft of this work, and criticizes the inefficiency of CTR-DRBG's update function. We stress that our modelling of the attack scenario and systematic treatment of how the issue affects each of the NIST DRBGs is novel.) This may not seem unreasonable if the DRBG produces only a handful of blocks per request; however since the standard allows for up to 2^{19} bits of output to be requested in each next call, there are situations in which the possibility of a partial state compromise occurring *during* output generation is worth considering.

Attack scenario: side channels. We consider an attacker who learns some information about the state variables being computed on during output generation. However, we assume the attacker is *not* able to perform a full memory compromise by which they would learn e.g., the output blocks $r^1, \ldots, r^n$ buffered in the internal memory, and in which case the security of all output in the call is lost.

A key attack vector by which an attacker might compromise such information is via a *side channel attack*. Notice that when multiple output blocks are generated in a single next call, there is a significant amount of computation going on 'under the hood' of the algorithm. For instance,

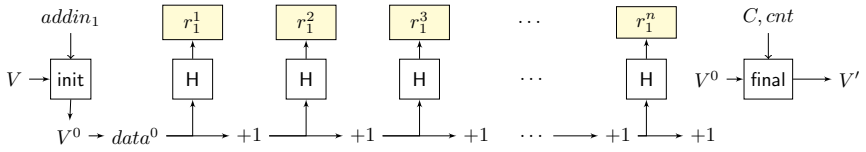

Fig. 7: Evolution of state $S = (V, C, cnt)$ within a next call of HASH-DRBG.

using CTR-DRBG with AES-128 to generate the maximum 2^{19} bits of output corresponds to $2^{12} = 4096$ AES-128 computations using a single key K^0, which remains constant through the iterative output generation process. Given that it is well known that AES invites leaky implementations [6, 32, 9, 28, 30, 26], to assume that such an attack will never be executed may be rather optimistic. Since forward security and robustness only allow the attacker to compromise the state *after* it has 'properly' updated (via the final process) at the conclusion of a next call, analysis with respect to these models tells us nothing about the impact of this.

Use case: buffering output. Bernstein [7] raises an important point about efficiency. Due to the extra block cipher calls incurred by the update process used to update the state of CTR-DRBG at the conclusion of the call (see right-hand column of Figure 6), an appealing usage choice is to generate a large output upfront in a single request, and buffer it to later be used for different purposes. Indeed, the SP 800-90A standard says of the performance of CTR-DRBG: "For large generate requests, CTR-DRBG produces outputs at the same speed as the underlying block cipher algorithm encrypts data", thereby highlighting the efficiency of this approach. The case is similar for HMAC-DRBG and HASH-DRBG.

Our attack model is intended to investigate the soundness of this buffering approach for scenarios in which partial state compromise during output generation via a side channel — which can only be exacerbated by such usage — is a realistic concern. Notice that when a DRBG is used in this way, some portions of the buffered output may be used for public values such as nonces whereas other portions of the output from the same call may be used for e.g., secret keys. To reflect this, our model assumes that the attacker learns an output block sent in the clear as e.g., a nonce, in conjunction to the partial state information gleaned via a side channel. The attacker's goal is to recover *unseen* output blocks used as security critical secrets, breaking the security of the consuming application.

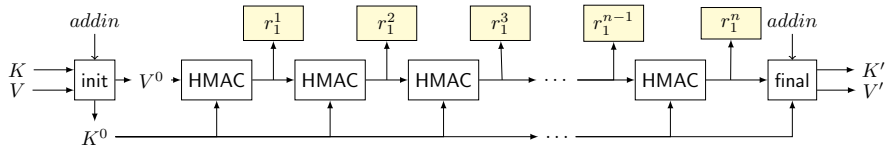

Fig. 8: Evolution of state (K, V, cnt) within a next call of HMAC-DRBG.

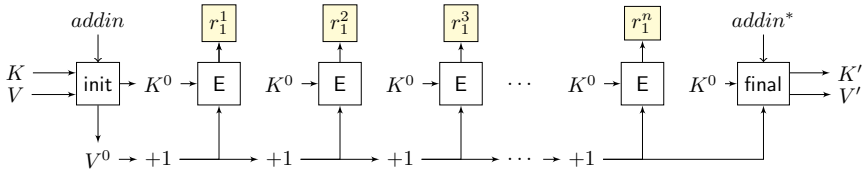

Fig. 9: Evolution of state $S = (K, V, cnt)$ within a next call of CTR-DRBG. Here $addin^* = addin$ if $addin \neq \varepsilon$ and $0^{\kappa+\ell}$ otherwise.

8.1 Attack Model

We now describe our attack model. Our presentation is intentionally fairly informal, and is intended to demonstrate key potential attacks rather than being an exhaustive treatment. We found that a more formal and / or code-based model using abstract leakage functions (in line with the literature on leakage-resilient cryptography e.g., [29, 18, 1]), added significant complexity to the exposition, without clarifying the presentation of the attacks or providing further insight. We therefore opted for a more informal written definition of the attack model which is nonetheless sufficiently precise to capture e.g., exactly what the attacker may learn, what he is challenged to guess, and so on.

Attack setup and goals. Consider Figure 5 depicting the structure of a next call for the NIST DRBGs decomposed into the subroutines $\mathcal{C} = (\mathsf{init}, \mathsf{gen}, \mathsf{final})$. Letting S denote the state at the start of the call, then this process defines a series of intermediate states / output blocks generated 'under the hood' of next during the course of the request:

$$(S, (S^0, data^0), r^1, (S^1, data^1), \ldots, r^n, (S^n, data^n), S') \ ,$$

with the algorithm finally returning $(R, S') = (r^1 \,\|\, \ldots \,\|\, r^m, S')$. (For simplicity, we assume the requested number of bits is a multiple of the block length; it is straightforward to generalize the model to remove this assumption.)

We consider an attacker $\mathcal{A}$ who is able to compromise a given component of an arbitrary intermediate state S^i (or in the case of HASH-DRBG, the additional state information $data^i$) for $i \in [0, n]$, in addition to an arbitrary output block r^j for $j \in [1, n]$ produced in the same call. We assume the indices (i, j) are known to $\mathcal{A}$[8]. We then assess the attacker's ability to achieve each of the following 'goals':

- **(1)** Recover unseen output blocks produced prior to the compromised block within the call $\{r^k\}_{k<j}$;
- **(2)** Recover unseen output blocks produced following the compromised block within the call $\{r^k\}_{k>j}$; and
- **(3)** Recover the state S' as updated at the conclusion of the call. This allows the attacker to run the generator forwards and recover future output.

	(1) Past output within call	**(2)** Future output within call	**(3)** Updated state S'	Additional input
CTR-DRBG // compromised K	✓	✓	✓	✓*
HMAC-DRBG // compromised K	×	✓	✓	×
HASH-DRBG // compromised V	✓	✓	×**	×

Fig. 10: Table summarizing our analysis. The leftmost three columns correspond to Section 8.1. The rightmost column corresponds to Section 8.5. A ✓ indicates that we demonstrate an attack. A × indicates that we believe the DRBG is not vulnerable to such an attack, with justification given. * corresponds to an attack if CTR-DRBG is implemented without a derivation function. ** indicates an exception in the case that $cnt = 1$ at the point of compromise.

Extended attack window if additional input not used. If additional input is not used in a next call, then init returns the state unchanged for each of the NIST DRBGs, corresponding to $S^0 = S$ in the above exposition. In this case, all attacks which apply when S^0 is compromised in our model can also be executed by an attacker who compromises the relevant component of the state S *prior* to the next call. This creates a greater window of opportunity in which this state may be compromised, as it will be set in memory following the conclusion of the previous call. As

[8] Here we assume the portion of public output contains a full output block, for which the attacker knows the index. This is a reasonable assumption, given that a TLS client or server random will contain at least one whole block, and at least 12 bytes of a second block (if 4 bytes of timestamp are used). Moreover, these values would be generated early in a call to the DRBG, and so have a low index j. However, both assumptions can be relaxed at the cost of the attacker performing more work to brute-force any missing bits and / or the index.

we shall see, not using additional input in a next call simplifies all attacks described, making state compromise in this case especially troubling.

Security analysis. We analyzed each of the NIST DRBGS with respect to our attack model, and summarize the key weaknesses found in Figure 10. We found that each of the NIST DRBGs exhibited vulnerabilities, with CTR-DRBG faring especially badly.

8.2 Security of CTR-DRBG with a Compromised Key

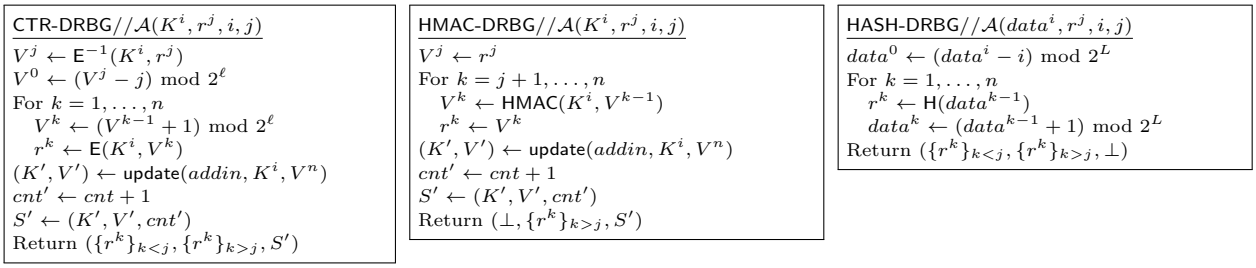

Fig. 11: Adversaries for attacks in Section 8.1.

The invertibility of the block cipher used by CTR-DRBG – and the fact that each output block is an encryption of the secret counter V — makes leakage of the secret key component of the state especially damaging.

Attack against CTR-DRBG with a compromised key. Consider the attacker $\mathcal{A}$ shown in the left-hand panel of Figure 11. We claim that for all $i \in [0, n]$ and $j \in [1, n]$, if additional input is not used ($addin = \varepsilon$) then $\mathcal{A}$ achieves goals **(1)**,**(2)** (recovery of all unseen output blocks produced in the next call) and **(3)** (recovery of the next state S') with probability one. If additional input is used ($addin \neq \varepsilon$) then the same statement holds for **(1)**, **(2)**, and the attacker's ability to satisfy **(3)** is equal to his ability to guess $addin$.

To see this, notice that each block of output produced in the next call is computed as $r^k = \mathsf{E}(K^0, V^0 + k)$ for $k \in [1, n]$, where K^0, V^0 denote the key and counter as returned by init at the start of output generation. Since the key does not update through this process, it holds that whatever intermediate key K^i attacker $\mathcal{A}$ compromises, this is the key used for output generation. It is then trivial for $\mathcal{A}$ to decrypt the output block r^j received in his challenge to recover the secret counter, thereby possessing all security critical state variables. However if $addin \neq \varepsilon$, then $\mathcal{A}$ must guess this string in order to compute S'.

Discussion. The fact that the attack recovers all unseen output from a given next call is especially damaging, since high value output blocks used as e.g., secret keys will be recovered *irrespective* of their position relative to the public block learnt by the attacker. This increases the exploitability of the compromised CTR-DRBG. In comparison, the infamously backdoored DualEC-DRBG only allowed recovery of output produced *after* the block learnt by the attacker, impacting its practical exploitability [12].

8.3 Security of HMAC-DRBG with a Compromised Key

Attack against HMAC-DRBG with a compromised key. Consider the attacker $\mathcal{A}$ shown in the middle panel of Figure 11, who compromises the key component of an intermediate state of HMAC-DRBG. We claim that for all $i \in [0, n]$ and $j \in [1, n]$, if $addin = \varepsilon$ then $\mathcal{A}$ achieves goals **(2)** and **(3)** with probability one. If $addin \neq \varepsilon$ then the same statement holds for **(2)**, and the attacker's ability to satisfy **(3)** is equal to his ability to guess $addin$.

To see this, let K^0, V^0 denote the state variables at the beginning of output generation. Output blocks are iteratively produced by computing $r^k = \mathsf{HMAC}(K^0, V^{k-1})$ for $k \in [1, n]$, and setting

$r^k = V^k$. Since the key does not update during this process, the key K^i compromised by the attacker will be equal to the key K^0 used for output generation. Since the output block r^j which $\mathcal{A}$ receives in his challenge is equal to the secret counter V^j, $\mathcal{A}$ now knows all security critical state variables of intermediate state S^j. $\mathcal{A}$ can then run HMAC-DRBG forward to recover all output produced following the compromised block in the call, and the updated state S' (subject to guessing *addin*).

Security of past output in a compromised next call. On a more positive note, it would appear that even if an attacker learns the entirety of an intermediate state S^i for $i \in [0, n]$ in addition to an output block r^j for $j \in [1, n]$, then it is still infeasible to achieve goal **(1)** and recover the set of output blocks $\{r^k\}_{k<j}$ produced prior to the compromised block within the call.

To see this, let V^0 denote the value of the counter at the start of output generation. Notice that for each $j \in [1, n]$, output block r^j takes the form:

$$r^j = V^j = \mathsf{HMAC}^j(K^0, V^0) \,,$$

where $\mathsf{HMAC}^i(K, \cdot)$ denotes the i^{th} iterate of $\mathsf{HMAC}(K, \cdot)$. As such, recovering prior blocks r^k for $k < j$ given K^0 and V^j corresponds to finding preimages of $\mathsf{HMAC}(K^0, \cdot)$. Since the key is known to the attacker, we clearly cannot argue that this is difficult based on the PRF-security of HMAC. However, modeling HMAC as a random oracle (Section 7), it follows that inverting HMAC for sufficiently high entropy V^0 is infeasible. Formalizing this intuition under a standard model assumption remains an interesting open question.

8.4 Security of HASH-DRBG with a compromised counter.

It is straightforward to see that if $\mathcal{A}$ learns the counter value V^i in the HASH-DRBG state, or the iterating copy of the counter in $data^i$ for any $i \in [0, n]$, $j \in [1, n]$, then $\mathcal{A}$ achieves goals **(1)** and **(2)** with probability one. Moreover, knowledge of the counter is sufficient to execute the attack; no output block is needed. The case in which $data^i$ is compromised is shown in the rightmost panel of Figure 11. However, unlike CTR-DRBG and HMAC-DRBG, without also learning the constant C goal **(3)** does not seem to be possible in general. We omit the details here due to space constraints, and provide a full discussion in Appendix C.

8.5 Security of Additional Input

We present an additional attack against an implementation of CTR-DRBG which does not use a derivation function. This attack can (under certain conditions) allow an attacker who compromises the state of the DRBG to also recover the strings of additional input fed to the DRBG during output generation requests. This is particularly concerning given that the standard allows these strings to contain secrets and sensitive data, provided they are not protected at a higher security strength than the instantiation.

Use of a derivation function. Consider the CTR-DRBG next algorithm (Section 3). If the derivation function is not used and additional input is included in a call, then the raw string of input is XORed directly into the CTR-DRBG state during the application of the update function (lines 8 and 15). One can verify from the pseudocode description of CTR-DRBG_df in Appendix G that to derive a $(\kappa + \ell)$-length string from a T-bit input requires N block cipher computations where $N = \lceil (\kappa + \ell)/\ell \rceil \cdot (\lceil (T + 72)/\ell \rceil + 2)$, and κ and ℓ denote the key and block size of the block cipher respectively. This computation is required for every next call which includes additional input, on top of each reseed and the initial state generation, and so represents a significant overhead[9].

Recovery of additional input. We describe the attack with respect to the 'ideal' conditions. Take an implementation of CTR-DRBG instantiated with AES-128 in which a derivation function is not used (the case for other approved block ciphers is totally analogous). Suppose that an attacker $\mathcal{A}$ has compromised the internal state $S = (K, V, cnt)$[10], and that the state compromise is followed

[9] A set of slides on the NIST DRBGs by Kelsey from 2004 [25] includes the comment "Block cipher derivation function is expensive and complicated... When gate count or code size is an issue, nice to be able to avoid using it!"

[10] Here we mean the usual definition of PRNG state, as opposed to the 'intermediate' states considered in the previous section.

by a next call in which additional input *addin* is used. Moreover, suppose *addin* has the form $addin = X_1 \| X_2$ where $X_1 \in \{0,1\}^{128}$ is known to the attacker and $X_2 \in \{0,1\}^{128}$ consists of 128 unknown bits. We assume X_2 includes a secret value such as a password which will be the target of the attack.

At the start of the next call, the state components K, V are updated with *addin* via $(K^0, V^0) \leftarrow \mathsf{update}(addin, K, V)$. It is straightforward to verify that

$$K^0 \| V^0 = K^* \| V^* \oplus addin = (K^* \oplus X_1) \| (V^* \oplus X_2) \ ,$$

where $K^* \| V^* = \mathsf{E}(K, V+1) \| \mathsf{E}(K, V+2)$. Since $\mathcal{A}$ has compromised (K, V), they can compute (K^*, V^*). Moreover, since X_1 is known to $\mathcal{A}$, it follows that the updated key $K^0 = (K^* \oplus X_1)$ is known to $\mathcal{A}$ also.

During output generation, output blocks are produced by encrypting the iterating counter under K^0. Therefore, the k^{th} block of output is of the form:

$$r^k = \mathsf{E}(K^0, V^0 + k) = \mathsf{E}(\mathbf{K^0}, (\mathbf{V}^* \oplus X_2) + \mathbf{k}) \ ,$$

where the variables in bold are known to $\mathcal{A}$. As such, each block of output produced is effectively an encryption of the target secret X_2 under a known key.

Given a single block of output r^k, $\mathcal{A}$ can *instantly* recover the target secret X_2 — consisting of 128-bits of unknown and secret data — as $X_2 = (\mathsf{E}^{-1}(K^0, r^k) - k) \oplus V^*$. Moreover, it is straightforward to verify that $\mathcal{A}$ has sufficient information to compute the state as updated following the next call. As such, $\mathcal{A}$ can continue to execute the same attack against subsequent output generation requests for as long as the key component of the state evolves predictably.

Extensions. In Appendix C we describe how to extend the attack to more general cases, and discuss how use of the derivation function prevents it.

9 Open Source Implementation Analysis

In Section 8, we showed that certain implementation decisions — permitted by the overly flexible standard — may influence the security guarantees of the NIST DRBGs. To determine if these decisions are taken by implementers in the real world, we investigated two open source implementations of CTR-DRBG, in OpenSSL [34] and mbed TLS [10]. We found that between the two libraries these problematic decisions have indeed been made.

Large output requests. As detailed in Section 8, generating many blocks of output in a single request increases both the likelihood and impact of our attacks. In OpenSSL, the next call of CTR-DRBG is implemented in the function `drbg_ctr_generate` in the file `drbg_ctr.c`. Interestingly — and contrary to the standard — this function does not impose *any limit* on the amount of random bits which may be requested per call. As such, an arbitrarily large output may be generated using a single key, exacerbating the attacks of Section 8.2. More generally, exceeding the output generation limit increases the likelihood of the well-known distinguishing attack against a block cipher in CTR-mode which uses colliding blocks to determine if an output is truly random.

By comparison, the implementation of CTR-DRBG in mbed TLS limits the number of output blocks per next call to 64 blocks of 128-bits. In the context of our attacks, this is much better for security than the 4,096 blocks allowed by the standard. Also this implementation forces a reseed after 10,000 calls to next, which is substantially lower than the 2^{48} calls that are allowed by the standard.

Derivation function. In Section 8.5, we showed that choosing to implement CTR-DRBG without the derivation function may allow the attacker to recover potentially sensitive data fed to the DRBG in the event of state compromise. We found that the OpenSSL implementation of CTR-DRBG allows the generator to be called simultaneously without the derivation function and with additional input. Specifically, by setting the flags field of the `RAND_DRBG_FLAG_CTR_NO_DF` structure to `RAND_DRBG_FLAG_CTR` the caller may suppress calls to the derivation function, presumably for performance purposes. As such, the attack described in Section 8.5 may be possible in real world implementations.

Summary. Despite the high level and theoretical nature of our analysis, an investigation of real world implementations shows that the problematic implementation decisions which we highlight

here *are* in fact decisions that implementers may make. While none of these decisions leads to an immediate vulnerability, both the implementation and usage of the functions may exacerbate other problems such as side channel or state compromise attacks. We hope that highlighting these issues will help implementers make informed decisions about how best to use these algorithms in the context of their implementation.

10 Conclusion

We conducted an in-depth and multi-layered analysis of the NIST SP 800-90A standard, with a focus on investigating unproven security claims and exploring flexibilities in the standard. On the positive side, we formally verify a number of the claimed — and yet, until now unproven — security properties in the standard. However, we argue that taking certain implementation choices permitted by the overly flexible standard may lead to vulnerabilities.

Design and prove simultaneously. Certain design features of the NIST DRBGs complicate their analysis, and a small tweak in design would facilitate a far simpler proof. This emphasizes the importance of developing cryptographic algorithms alongside security proofs, and — more importantly — not standardizing algorithms with unproven security properties.

Flexibility in algorithm specifications. The attacks of Section 8 are both facilitated, and exacerbated, by certain implementation choices allowed by the overly flexible standard. In Section 9, we confirmed that implementers do make these choices in the real world. This may be a warning to standard writers to avoid unnecessary flexibility, as it may lead to unintended vulnerabilities.

Usage recommendations. Fortunately, because these vulnerabilities stem from implementation choices, we may offer recommendations to make the use of these algorithms more secure. First off, if the algorithms are being run in a setting where side channel attacks are a concern then CTR-DRBG should not be used. Additional input should be (safely) incorporated during output generation wherever possible and the DRBG should be reseeded with fresh entropy as often as is practical. While the standard allows outputs of sizeable length to be requested, users should not 'batch up' calls by making a single call for all randomness required for an application and separating this into separate values. Finally, the CTR-DRBG derivation function should always be used.

Future work. Analyzing the robustness of CTR-DRBG is an important direction for further work. The key challenge with this seems to be proving that the derivation function is a good randomness extractor. Unfortunately, since the underlying CBC-MAC-based extractor (BCC in Appendix G) is applied multiple times to the same entropy sample, existing results on the extraction properties of CBC-MAC [15, 38] cannot be applied. A proof of robustness in the IPM [20] may be possible. More generally, the design flexibilities we critique above are related to efficiency savings. PRNG design which achieves an optimal balance between security and efficiency is a key direction for future work. For example, redesigning the CTR-DRBG derivation function to avoid computational overhead would make its use more palatable. The gap between the specification of these DRBGs which allows for various optional inputs and implementation choices, and the far simpler manner in which PRNGs are typically modeled in the literature could indicate that theoretical models are not adequately capturing real world PRNGs. Extending these models, may help understand the limits and possibilities of what can be achieved.

Acknowledgements. The authors thank the anonymous reviewers whose many insightful comments and suggestions have greatly improved this paper.

Bibliography

[1] Michel Abdalla, Sonia Belaïd, David Pointcheval, Sylvain Ruhault, and Damien Vergnaud. Robust pseudo-random number generators with input secure against side-channel attacks. In *ACNS*, 2015.

[2] E Barker and J Kelsey. Nist sp 800-90a rev. 1 recommendation for random number generation using deterministic random bit generators. *Retrieved September*, 3:2016, 2015.

[3] Elaine Barker and John Kelsey. Draft nist sp 800-90c. recommendation for random bit generator (rbg) constructions. 2012.
[4] Mihir Bellare, Thomas Ristenpart, and Stefano Tessaro. Multi-instance security and its application to password-based cryptography. In *CRYPTO*. 2012.
[5] Mihir Bellare and Bennet Yee. Forward-security in private-key cryptography. In *CT-RSA*, 2003.
[6] Daniel J Bernstein. Cache-timing attacks on aes, 2005.
[7] Daniel J. Bernstein. Fast-key-erasure random-number-generators, 2017.
[8] Daniel J Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko Van Someren. Factoring RSA keys from certified smart cards: Coppersmith in the wild. In *ASIACRYPT*, 2013.
[9] Andrey Bogdanov. Improved side-channel collision attacks on aes. In *Selected Areas in Cryptography*, volume 4876, pages 84–95. Springer, 2007.
[10] Simon Butcher, Janos Follath, and Andrés Amaya García. mbed tls. https://tls.mbed.org/, 2015-2018.
[11] Matthew J Campagna. Security bounds for the nist codebook-based deterministic random bit generator. *IACR Cryptology ePrint Archive*, 2006:379, 2006.
[12] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J Bernstein, Jake Maskiewicz, Hovav Shacham, Matthew Fredrikson, et al. On the practical exploitability of dual ec in tls implementations. In *USENIX*, 2014.
[13] Shan Chen and John Steinberger. Tight security bounds for key-alternating ciphers. In *CRYPTO*, 2014.
[14] Mario Cornejo and Sylvain Ruhault. Characterization of real-life prngs under partial state corruption. In *ACM CCS*, 2014.
[15] Yevgeniy Dodis, Rosario Gennaro, Johan Håstad, Hugo Krawczyk, and Tal Rabin. Randomness extraction and key derivation using the cbc, cascade and hmac modes. In *Advances in Cryptology–CRYPTO 2004*, pages 115–133. Springer, 2004.
[16] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input:/dev/random is not robust. In *ACM CCS*, 2013.
[17] Yevgeniy Dodis, Thomas Ristenpart, John P Steinberger, and Stefano Tessaro. To hash or not to hash again?(in) differentiability results for h 2 and hmac. In *CRYPTO*, 2012.
[18] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 293–302. IEEE, 2008.
[19] PUB FIPS. 140-2. *Security Requirements for Cryptographic Modules*, 25, 2001.
[20] Peter Gaži and Stefano Tessaro. Provably robust sponge-based prngs and kdfs. In *CRYPTO*, 2016.
[21] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *USENIX*, 2012.
[22] Shoichi Hirose. Security analysis of drbg using hmac in nist sp 800-90. In *WISA*, 2008.
[23] Wilson Kan. Analysis of underlying assumptions in NIST DRBGs. *eprint*, 2007.
[24] Q Ye Katherine, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W Appel. Verified correctness and security of mbedtls hmac-drbg. *ACM CCS*, 2017.
[25] J. Kelsey. Five drbg algorithms based on hash functions & block ciphers, 2004.
[26] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.
[27] Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. In *CRYPTO*, 2010.
[28] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *International Conference on Information Security and Cryptology*, pages 343–358. Springer, 2002.
[29] Silvio Micali and Leonid Reyzin. Physically observable cryptography. In *Theory of Cryptography Conference*, pages 278–296. Springer, 2004.
[30] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[31] Jacques Patarin. The "coefficients h" technique. In *SAC*, 2008.
[32] Colin Percival. Cache missing for fun and profit, 2005.
[33] Nicole Perlroth. Government announces steps to restore confidence on encryption standards. 2013.
[34] The OpenSSL Project. Openssl. https://www.openssl.org/, 1998-2018.
[35] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 487–506. Springer, 2011.
[36] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.
[37] Sylvain Ruhault. Sok: Security models for pseudo-random number generators. *IACR Transactions on Symmetric Cryptology*, 2017.
[38] Thomas Shrimpton and R Seth Terashima. A provable security analysis of intel's secure key RNG. In *EUROCRYPT*, 2015.
[39] Thomas Shrimpton and R Seth Terashima. Salvaging weak security bounds for blockcipher-based constructions. In *ASIACRYPT*, 2016.
[40] Dan Shumow and Niels Ferguson. On the possibility of a back door in the nist sp800-90 dual ec prng. In *CRYPTO rump session*, 2007.
[41] Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry A. McKay, Mary L. Baish, and Mike Boyle. Sp 800-90b. recommendation for the entropy sources used for random bit generation. 2012.
[42] Apostol Vassilev and Willy May. Annex c: Approved random number generators for fips pub 140-2, security requirements for cryptographic modules. 2016.
[43] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 debian openssl vulnerability. In *ACM SIGCOMM*, 2009.

A Definitions and Discusssion from Section 2

Entropy. For distributions X and Z over sets $\mathcal{X}$ and $\mathcal{Z}$ respectively, we define the *worst-case min-entropy* of X conditioned on Z to be

$$\mathrm{H}_\infty(X|Z) = -\log\left(\max_{x\in\mathcal{X},z\in\mathcal{Z}} \Pr[\, X = x \mid Z = z \,]\right),$$

and define the *average-case min-entropy* of X conditioned on Z as

$$\tilde{\mathrm{H}}_\infty(X|Z) = -\log\left(\sum_{z\in\mathcal{Z}} \max_{x\in\mathcal{X}} \Pr[\, X = x \mid Z = z \,]\cdot\Pr[\, Z = z \,]\right).$$

Cryptographic components. We let $\mathrm{Func}(Dom, Rng)$ denote the set of all functions $f : Dom \to Rng$. The pseudorandom function (PRF) distinguishing advantage of an adversary $\mathcal{A}$ against a keyed function $\mathsf{F} : Keys \times Dom \to Rng$ given q oracle queries is defined

$$\mathsf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(\mathcal{A}, q) = |\Pr\left[\,\mathcal{A}^{\mathsf{F}(K,\cdot)} \Rightarrow 1 :\ K \xleftarrow{\$} \{0,1\}^\kappa\,\right] - \Pr\left[\,\mathcal{A}^{\pi(\cdot)} \Rightarrow 1 : \pi \xleftarrow{\$} \mathrm{Func}(Dom, Rng)\,\right]|,$$

where the superscript denotes a functionality that $\mathcal{A}$ is given oracle access to. A function $\mathsf{E} : \{0,1\}^\kappa \times \{0,1\}^\ell \to \{0,1\}^\ell$ is called a block cipher if $\mathsf{E}(K,\cdot)$ is a permutation on $\{0,1\}^\ell$ for each $K \in \{0,1\}^\kappa$. We let $\mathsf{D}(K,\cdot)$ denote the inverse (or decryption) of $\mathsf{E}(K,\cdot)$, so $\mathsf{D}(K,\mathsf{E}(K,m)) = m$ for all $m \in \{0,1\}^\ell, K \in \{0,1\}^\kappa$.

The H-Coefficient technique. Patarin's H-coefficient technique [31] has proved to be a highly useful tool for analyzing indistinguishability experiments. We recall the formulation of this method from [13] below, and refer the reader to that work for a full discussion of the approach. The proof technique consider two experiments, which we denote Real and Ideal, and a deterministic and computationally unbounded adversary $\mathcal{A}$ who tries to distinguish the two. We define a *transcript* which captures $\mathcal{A}$'s view of the experiments, say that a transcript is *valid* if it could be produced by an execution of one of the experiments, and let T_0 and T_1 denote the distributions of valid transcripts corresponding to the real and ideal experiments respectively. We additionally define a set of Bad transcripts, and view all other valid transcripts as being Good. The following theorem then allows us to bound the advantage $\mathcal{A}$ has in distinguishing the real and ideal experiments as follows:

Theorem 5. *Suppose that there exist $\delta, \epsilon \in [0,1]$ such that for all transcripts $\tau \in \mathsf{Good}$ it holds that*

$$\Pr[\,\mathsf{T}_0 = \tau\,]/\Pr[\,\mathsf{T}_1 = \tau\,] \geq 1 - \epsilon,$$

and moreover it holds that $\Pr[\,\mathsf{T}_1 \in \mathsf{bad}\,] \leq \delta$. Then the probability

$$|\Pr[\,\mathcal{A} \Rightarrow 1 \text{ in } \mathsf{Real}\,] - \Pr\mathcal{A} \Rightarrow 1 \text{ in } \mathsf{Ideal} \leq \epsilon + \delta.$$

Discussion of security model. Our PRNG definition (Definition 1) follows that of Dodis et al. [16], with a number of modifications. Following Shrimpton et al. [38], we define setup to be the algorithm which constructs the initial state of the PRNG from samples drawn from the entropy source. We assume that the salt X is generated externally and supplied to the PRNG. In contrast, in [16] the initial state is defined to be a uniform bit string which is supplied to the PRNG, and setup is used to generate the salt. These modifications allow us to better model real-world PRNGs which must construct their initial state from the entropy source. We further modify the syntax of setup from [38] to have it take an entropy sample and nonce as input, as per the specification of the NIST DRBGs. In contrast in [38], setup takes no input, but may have access to the entropy source. Finally, we extend the definitions of $\mathsf{refresh}$ and next from [16] to allow: **(1)** entropy samples

I to be bit strings of arbitrary length $|I| \in [\alpha_{min}, \alpha_{max}]$, as opposed to some fixed length α_{max}; **(2)** variable length outputs up to length α_{out} to be requested via the parameter β; and **(3)** the option to include strings of additional input $addin \in \{0,1\}^{\leq \alpha_{add}}$ in next calls. As we shall see in Section 3, this is necessary to capture the interface of the corresponding algorithms for the NIST DRBGs.

B Discussion on Section 3.

In this section, we offer further discussion of the standard, which was omitted from Section 3 due to space constraints.

The counter of HASH-DRBG. The constant C is added into the counter V during each state update, but is itself only updated following a reseed. The standard does not explicitly state the purpose of C; however slides by Kelsey from 2004 [25] mention how HASH-DRBG will "Hash with constant to avoid duplicating other hash computations". As such, it would appear that the purpose of the constant is to ensure that if a previous counter V is duplicated at some point in a different refresh period, then the inclusion of the (almost certainly distinct) counter in the subsequent state update prevents the previous sequence of states being repeated.

C Discussion from Section 8

Security of the updated state. As discussed in Section 8.4, learning V^i or $data^i$ for any $i \in [0, n]$ is sufficient to compute all output produced in the call, by allowing the attacker to recover the value of the initial counter V^0 which was hashed to produce the output blocks. However, interestingly — and unlike the other NIST DRBGs — this is insufficient to recover the updated state S' in general. At the end of the next call, the new counter is computed as

$$V' = (V^0 + \mathsf{H}(0\text{x}03||V^0) + C + cnt) \bmod 2^L \ ;$$

however, for all but the first next call, it seems to be infeasible to extract the constant C from the known counter V^0 without inverting the hash function. (The exception with the first next call is because C is derived deterministically from the initial counter V during the setup process. Provided additional input is not used in the call, this is the counter $\mathcal{A}$ will be able to recover during the attack, allowing them to compute C themselves.) That said, if additional input is not used and an attacker compromises counters V^0 and $V^{0\prime}$ used for output generation from two consecutive next calls, then $\mathcal{A}$ can easily recover C by calculating

$$C = (V^{0\prime} - \mathsf{H}(0\text{x}03||V^0) - (cnt + 1)) \bmod 2^L \ ,$$

where cnt denotes the reseed counter at the point of the call, thus facilitating the recovery of the updated state, and subsequent output. The same attack is possible if additional input is used conditional on $\mathcal{A}$ being able to guess its value.

C.1 Discussion from Section 8.5

Extensions to additional input recovery attack. In Section 8.5 we described the ideal conditions for the attack. However the attack is still possible if X_1 is not known to the attacker. Supposing X_1 has γ-bits of entropy, then repeating the above process for each possible candidate for X_1 will recover the correct secret X_2 among a list of 2^γ candidates. If the data in X_2 is of a distinctive structure, or multiple output blocks can be recovered from the compromised next call, this can help $\mathcal{A}$ quickly eliminate candidates. Either way, the entropy of X_2 is reduced to γ-bits, a loss which — given X_2 contains up to 128-bits of unknown data for the instantiations permitted by the standard — may be substantial.

Security benefit of the derivation function. The attack exploits the way in which the structure of $addin$ is preserved by XOR. When the derivation function is used to condition the data, this structure is sufficiently destroyed that the above attack no longer works. Indeed, even if the attacker

could compromise the raw derivation function output it is difficult to see how to recover the underlying additional input string more efficiently than by exhausting its entropy in a brute-force attack. Likewise for HASH-DRBG and HMAC-DRBG, in which additional input is hashed as it is incorporated into the state, it would appear that recovery of such strings requires a brute-force attack.

D Proofs from Section 6

Proof of Theorem 1. Let $(\mathcal{A}, \mathcal{D})$ be an attacker / sampler pair in game $\mathrm{Rob}^{\mathcal{D},\mathcal{A}}_{\mathcal{G},\gamma^*,\beta}$ against $\mathcal{G}$. We shall construct a hybrid argument based upon the attacker's RoR queries, where by assumption $\mathcal{A}$ makes q_R such queries. We say that a RoR query is uncompromised if $c \geq \gamma^*$ at the point of the query; otherwise we say it is compromised. We further divide uncompromised RoR queries into those which are *preserving* and those which are *recovering*. We say a RoR query is *recovering* if $c < \gamma^*$ at some point between the previous RoR query (or if there have been no previous RoR queries, the start of the game) and the current one inclusive. For a recovering RoR query, we call the most recent query for which $c \leftarrow 0$ the most recent entropy drain (mRED) query. If an uncompromised query is not recovering, then it is said to be *preserving*.

Let game G^*_0 be equivalent to game $\mathrm{Rob}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\gamma^*,\beta}$ with challenge bit $b = 0$. Let $(\mathcal{A}, \mathcal{D})$ be an attacker / sampler pair in G^*_0. We begin by defining game G_0, which is identical to G^*_0 except we replace the line $S \leftarrow \mathsf{setup}^{\mathsf{H}}(X, I, N)$ with $S \leftarrow\!\!\$\ \mathsf{M}^{\mathsf{H}}(\varepsilon)$. We bound the gap between these games with a reduction to the Init security of $\mathcal{G}$. Let $(\mathcal{A}_1, \mathcal{D})$ be an attacker / sampler pair in game Init, where $\mathcal{A}_1$ proceeds as follows. When $\mathcal{A}_1$ receives his challenge state S^* and associated entropy samples, estimates, side information, and nonce $((I_i)_{i=2}^{q_{\mathcal{D}}+1}, (\gamma_i, z_i)_{i=1}^{q_{\mathcal{D}}+1}, N)$, he passes $(X, (\gamma, z, N))$ to $\mathcal{A}$. $\mathcal{A}_1$ begins to simulate game Rob with challenge bit $b = 0$ for $\mathcal{A}$, using the challenge state S^* as the initial state for the simulated game. In particular, $\mathcal{A}_1$ uses the remaining entropy samples and estimates to simulate $\mathcal{A}$'s Ref calls and uses his own H oracle to answer $\mathcal{A}$'s H queries and to simulate the $\mathsf{refresh}^{\mathsf{H}}$ and $\mathsf{next}^{\mathsf{H}}$ algorithms. At the end of the game, $\mathcal{A}_1$ outputs whatever bit $\mathcal{A}$ does. Notice that if $\mathcal{A}$ received a real state in his challenge then this perfectly simulates G^*_0; otherwise it perfectly simulates G_0. Moreover, notice that $\mathcal{A}_1$ makes at most $(q_{\mathsf{H}} + q_R \cdot q_{\mathrm{nxt}} + q_{\mathcal{D}} \cdot q_{\mathrm{ref}})$ queries to H (this total follows from the q_{H} queries which $\mathcal{A}$ makes to H and which $\mathcal{A}_1$ forwards to his own oracle, plus the q_{ref} (resp. q_{nxt}) H queries which $\mathcal{A}_1$ makes to simulate the $q_{\mathcal{D}}$ (resp. q_R) Ref (resp. RoR) queries. It follows that

$$|\Pr[\,\mathcal{A} \Rightarrow 1 \text{ in } G^*_0\,] - \Pr[\,\mathcal{A} \Rightarrow 1 \text{ in } G_0\,]| \leq \mathrm{Adv}^{\mathrm{init}}_{\mathcal{G},\mathsf{M},\gamma^*}(\mathcal{A}_1, \mathcal{D}) \ .$$

We now define a series of modified games, where for each $i \in [0, q_R - 1]$, G_{i+1} is identical to game G_i except in the event that the $(i+1)^{\mathrm{st}}$ RoR query is uncompromised. In this case, then instead of computing the output / state via $(R, S') \leftarrow \mathsf{next}^{\mathsf{H}}(X, S, \beta, addin)$ where S denotes the current state of the PRNG the challenger instead returns $R \leftarrow\!\!\$\ \{0,1\}^{\beta}$ to $\mathcal{A}$ regardless of the challenge bit and sets the state of the PRNG to $\mathsf{M}^{\mathsf{H}}(S)$ (e.g., the mask of the state which was input to next to satisfy the output request). This new state is used to run the rest of the game. We also introduce an intermediate game $G_{(i+\frac{1}{2})}$ defined such that if the $(i+1)^{\mathrm{st}}$ RoR query is preserving then the challenger acts as in Game $(i+1)$, whereas if the $(i+1)^{\mathrm{st}}$ RoR query is recovering the challenger acts as in Game i. We bound the gaps between these pairs of games in the following lemma.

Lemma 7. *For any $(q_{\mathsf{H}}, q_{\mathcal{D}}, q_R, q_S)$-adversary $\mathcal{A}$ and sampler $\mathcal{D}$ and for all $i \in [0, q_R - 1]$, there exist $(q_{\mathsf{H}} + q_{\mathcal{D}} \cdot q_{ref} + q_R \cdot (q_{nxt} + q_{\mathsf{M}}))$-adversaries $\mathcal{A}^i_2$, $\mathcal{A}^i_3$ such that*

$$|\Pr[\,\mathcal{A} \Rightarrow 1 \text{ in } G_i\,] - \Pr\left[\,\mathcal{A} \Rightarrow 1 \text{ in } G_{(i+1)}\,\right]| \leq \mathrm{Adv}^{\mathrm{pres}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}^i_2) + \mathrm{Adv}^{\mathrm{rec}}_{\mathcal{G},\mathsf{M},\beta,\gamma^*}(\mathcal{A}^i_3, \mathcal{D}) \ .$$

Proof: For each $i \in [0, q_R - 1]$, the triangle inequality implies that

$$\begin{aligned}
&|\Pr[\,\mathcal{A} \Rightarrow 1 \text{ in } G_i\,] - \Pr\left[\,\mathcal{A} \Rightarrow 1 \text{ in } G_{(i+1)}\,\right]| \\
&\quad \leq |\Pr[\,\mathcal{A} \Rightarrow 1 \text{ in } G_i\,] - \Pr\left[\,\mathcal{A} \Rightarrow 1 \text{ in } G_{(i+\frac{1}{2})}\,\right]| \\
&\qquad + |\Pr\left[\,\mathcal{A} \Rightarrow 1 \text{ in } G_{(i+\frac{1}{2})}\,\right] - \Pr\left[\,\mathcal{A} \Rightarrow 1 \text{ in } G_{(i+1)}\,\right]| \ .
\end{aligned}$$

We begin by showing that there exists an adversary $\mathcal{A}_2^i$ such that

$$|\Pr[\mathcal{A} \Rightarrow 1 \text{ in } G_i] - \Pr\left[\mathcal{A} \Rightarrow 1 \text{ in } G_{(i+\frac{1}{2})}\right]| \leq \mathrm{Adv}^{\mathrm{pres}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}_2^i) .$$

We may assume without loss of generality that the $(i+1)^{\mathrm{st}}$ query is preserving, otherwise games G_i and are $G_{(i+\frac{1}{2})}$ are identical. Attacker $\mathcal{A}_2^i$ proceeds as follows. $\mathcal{A}_2^i$ runs $\mathcal{A}$ as a subroutine, using his H oracle as well as the code of the sampler $\mathcal{D}$ to simulate game G_i up to and including the i^{th} RoR query. In more detail: on input salt X, $\mathcal{A}_2^i$ first computes $(\sigma_1, I_1, \gamma_1, z_1) \leftarrow\!\!\$\ \mathcal{D}(\sigma_0)$, and passes $(X, (\gamma_1, z_1, N))$ to $\mathcal{A}$ where $N \leftarrow \mathcal{N}$. $\mathcal{A}_2^i$ simulates $\mathcal{A}$'s H oracle by querying his own H oracle and returning the response. For a Ref query, $\mathcal{A}_2^i$ uses the code of the sampler to compute $(\sigma', I, \gamma, z) \leftarrow\!\!\$\ \mathcal{D}(\sigma)$ and returns (γ, z) to $\mathcal{A}$, but does not yet update the state. For Get, Set, and RoR queries, $\mathcal{A}_2^i$ takes the state S which immediately followed the last non-Ref query (or sets $S = \mathsf{M}^{\mathsf{H}}(\varepsilon)$ if this is the first non-Ref query and so no such state exists) and uses $\mathsf{refresh}^{\mathsf{H}}$ to refresh that state with all entropy samples which would have been incorporated due to Ref queries since the previous call. ultimately producing state S'. For Get / Set queries, $\mathcal{A}_2^i$ returns / overwrites the state S' as required. For the first $(i-1)$ RoR queries, if the query is uncompromised then $\mathcal{A}_2^i$ samples $R \leftarrow\!\!\$\ \{0,1\}^\beta$ and sets the state of the generator to $S'' = \mathsf{M}^{\mathsf{H}}(S')$. If the query is compromised then $\mathcal{A}_2^i$ simply computes $(R, S'') \leftarrow \mathsf{next}^{\mathsf{H}}(X, S', \beta, addin)$. In both cases, $\mathcal{A}$ returns R to $\mathcal{A}$ and continues running the game with state S''. $\mathcal{A}$ simulates the i^{th} RoR query (which must be uncompromised, since we have assumed the $(i+1)^{\mathrm{st}}$ query is preserving) as just described, but does not mask the state S'. (Since the $(i+1)^{\mathrm{st}}$ RoR query is preserving, we know that there can be only Ref queries between the i^{th} and $(i+1)^{\mathrm{st}}$ RoR queries. Looking ahead, $\mathcal{A}_2^i$ will insert his challenge in the $(i+1)^{\mathrm{st}}$ RoR query, and S' will be masked as part of the challenge computation.)

To simulate the $(i+1)^{\mathrm{st}}$ RoR query with associated additional input $addin$, $\mathcal{A}_2^i$ passes the state S' which immediately followed the i^{th} RoR query (or $S' = \varepsilon$ if $i = 0$ and so no such state exists), as well as $addin$ and all entropy inputs $I_1, \ldots, I_d$ which were output by the simulated sampler in response to Ref queries made in between the i^{th} and $(i+1)^{\mathrm{st}}$ RoR queries, to his challenger. $\mathcal{A}_2^i$ receives (R^*, S^*) in response and returns R^* to $\mathcal{A}$. $\mathcal{A}_2^i$ continues running the game with state S^* and the simulated algorithms $\mathsf{refresh}^{\mathsf{H}}$ and $\mathsf{next}^{\mathsf{H}}$. At the end of the game, $\mathcal{A}_2^i$ outputs whatever bit $\mathcal{A}$ does.

Notice that if $\mathcal{A}_2^i$'s challenge bit is 0 and he receives the real output and state in his challenge then this perfectly simulates G_i; otherwise he receives a random output and a masked state, and this perfectly simulates game $G_{(i+\frac{1}{2})}$. It follows that

$$\begin{aligned}
&|\Pr[\mathcal{A} \Rightarrow 1 \text{ in } G_i] - \Pr\left[\mathcal{A} \Rightarrow 1 \text{ in } G_{(i+\frac{1}{2})}\right]| \\
&= |\Pr[\mathcal{A}_2^i \Rightarrow 1 \mid b = 0] - \Pr[\mathcal{A}_2^i \Rightarrow 1 \mid b = 1]| \leq \mathrm{Adv}^{\mathrm{pres}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}_2^i) .
\end{aligned}$$

Moreover, notice that $\mathcal{A}_2^i$ makes at most $(q_{\mathsf{H}} + q_{\mathcal{D}} \cdot q_{\mathrm{ref}} + q_R \cdot (q_{\mathrm{nxt}} + q_{\mathsf{M}}))$ queries to H. Here the additional $q_R \cdot q_M$ term arises since $\mathcal{A}_2^i$ masks at most $(q_R - 1)$ PRNG states (one for every uncompromised RoR query with index $j \in [1, i]$) plus the initial state $S_0 \leftarrow\!\!\$\ \mathsf{M}^{\mathsf{H}}(\varepsilon)$. The other terms arise from $\mathcal{A}_2^i$ answering $\mathcal{A}$'s H queries and simulating the $\mathsf{refresh}^{\mathsf{H}}$ and $\mathsf{next}^{\mathsf{H}}$ algorithms.

Next, we show that for all $i \in [0, q_R - 1]$ there exists an adversary $\mathcal{A}_3^i$ such that

$$|\Pr\left[\mathcal{A} \Rightarrow 1 \text{ in } G_{(i+\frac{1}{2})}\right] - \Pr[\mathcal{A} \Rightarrow 1 \text{ in } G_{(i+1)}]| \leq \mathrm{Adv}^{\mathrm{rec}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta}(\mathcal{A}_3^i, \mathcal{D}) .$$

We may again assume without loss of generality that the $(i+1)^{\mathrm{st}}$ query is recovering otherwise games $G_{(i+\frac{1}{2})}$ and $G_{(i+1)}$ are identical. Attacker $\mathcal{A}_3^i$ proceeds as follows. Attacker $\mathcal{A}_3^i$ is given salt X and $(I_1, (\gamma_i, z_i))_{i=1}^{q_{\mathcal{D}}+1}, N)$, passes the appropriate values to $\mathcal{A}$, and begins to simulate game Rob. $\mathcal{A}_3^i$ simulates $\mathcal{A}$'s oracles for the first i queries as described above with two differences. Firstly in response to a Ref query, $\mathcal{A}_3^i$ returns the relevant entropy estimate / side information (which were given to $\mathcal{A}_3^i$ at the start of his challenge) to $\mathcal{A}$. Then at the point of the next non-Ref query, $\mathcal{A}_3^i$ queries Sam repeatedly to get all entropy samples which should have been incorporated into the state during that period and uses these to update the state. (In contrast, in the previous simulation $\mathcal{A}_2^i$ ran the code of the sampler himself.) The second difference is that if the i^{th} RoR query is uncompromised, $\mathcal{A}$ returns $R \leftarrow\!\!\$\ \{0,1\}^\beta$ to $\mathcal{A}$ as before but now also sets the state to $S'' = \mathsf{M}^{\mathsf{H}}(S')$ where S' denotes the state of the PRNG as updated with entropy samples at the point of the query. (In the previous simulation $\mathcal{A}_2^i$ left this state unmasked.)

When $\mathcal{A}$ makes his $(i+1)^{\text{st}}$ RoR query with associated additional input *addin*, $\mathcal{A}_3^i$ locates the mRED query (which must exist since we have assumed the $(i+1)^{\text{st}}$ query is recovering). $\mathcal{A}_3^i$ submits the state which immediately followed the mRED query to his challenger, along with d set equal to the number of Ref calls which $\mathcal{A}$ made between the mRED query and the $(i+1)^{\text{st}}$ RoR query, and *addin*. $\mathcal{A}_3^i$ receives (R^*, S^*) in response along with the remaining entropy samples. $\mathcal{A}_3^i$ returns R^* to $\mathcal{A}$, and uses the state S^* along with the entropy samples to simulate the remainder of the game. At the end of the game $\mathcal{A}_3^i$ outputs whatever bit $\mathcal{A}$ does.

Notice that if $\mathcal{A}_3^i$'s challenge bit is 0 and so he receives the real output and state in his challenge then this perfectly simulates $G_{(i+\frac{1}{2})}$; otherwise, he receives a random output and a masked state and this perfectly simulates game $G_{(i+1)}$. It follows that

$$\begin{aligned}&|\Pr\left[\,\mathcal{A}\Rightarrow 1 \text{ in } G_{(i+\frac{1}{2})}\,\right] - \Pr\left[\,\mathcal{A}\Rightarrow 1 \text{ in } G_{(i+1)}\,\right]|\\ &\leq |\Pr\left[\,\mathcal{A}_3^i \Rightarrow 1 \mid b=0\,\right] - \Pr\left[\,\mathcal{A}_3^i \Rightarrow 1 \mid b=1\,\right] \leq \mathrm{Adv}^{\mathrm{rec}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta}(\mathcal{A}_3^i,\mathcal{D})\ ,\end{aligned}$$

where an analogous argument to that made above verifies the query complexity of the adversary.

Now in game G_{q_R} each one of $\mathcal{A}$'s uncompromised RoR queries is answered with a random bit string and a masked state. To return to using unmasked states while still returning random outputs to $\mathcal{A}$, we define a second sequence of hybrid games where game $\bar{G}_{q_R}$ is identical to game G_{q_R}, and for each $i \in [0, q_R-1]$ game $\bar{G}_i$ is identical to game $\bar{G}_{(i+1)}$ except if the $(i+1)^{\text{st}}$ query is uncompromised. In this case, for the $(i+1)^{\text{st}}$ query the challenger computes $(R, S'') \leftarrow \mathsf{next}^{\mathsf{H}}(X, S', \beta, addin)$ where S' denotes the state at the point of the query and returns $R \leftarrow\!\!\$\ \{0,1\}^\beta$ to the attacker, but continues running the game with the real state S'' as opposed to using the masked version of that state $\mathsf{M}^{\mathsf{H}}(S')$. We define intermediate games $\bar{G}_{(i+\frac{1}{2})}$ analogously to $G_{(i+\frac{1}{2})}$. An analogous argument to that used above then implies the following lemma.

Lemma 8. *For any $(q_{\mathsf{H}}, q_{\mathcal{D}}, q_R, q_S)$-adversary $\mathcal{A}$ and sampler $\mathcal{D}$ and for all $i \in [0, q_R-1]$, there exist $(q_{\mathsf{H}} + q_{\mathcal{D}} \cdot q_{ref} + q_R \cdot (q_{nxt} + q_{\mathsf{M}}))$-adversaries $\bar{\mathcal{A}}_2^i$, $\bar{\mathcal{A}}_3^i$ such that*

$$|\Pr\left[\,\mathcal{A}\Rightarrow 1 \text{ in } \bar{G}_i\,\right] - \Pr\left[\,\mathcal{A}\Rightarrow 1 \text{ in } \bar{G}_{(i+1)}\,\right]| \leq \mathrm{Adv}^{\mathrm{pres}}_{\mathcal{G},\mathsf{M},\beta}(\bar{\mathcal{A}}_2^i) + \mathrm{Adv}^{\mathrm{rec}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta}(\bar{\mathcal{A}}_3^i,\mathcal{D})\ .$$

The proof of the above lemma is identical to that in the previous case, except that now $\bar{\mathcal{A}}_2^i$ and $\bar{\mathcal{A}}_3^i$ return random outputs in response to all uncompromised RoR queries made by $\mathcal{A}$. Similarly, we let $\bar{G}_0^*$ be identical to game $\bar{G}_0$, except we compute the initial state via $S_0 \leftarrow \mathsf{setup}^{\mathsf{H}}(X, I_1, N)$ rather than as $\mathsf{M}^{\mathsf{H}}(\varepsilon)$. An analogous argument to that used above again implies that there exists an adversary $\bar{\mathcal{A}}_1$ making at most $(q_{\mathsf{H}} + q_R \cdot q_{\mathrm{nxt}} + q_{\mathcal{D}} \cdot q_{\mathrm{ref}})$ queries such that

$$|\Pr\left[\,\mathcal{A}\Rightarrow 1 \text{ in } \bar{G}_0\,\right] - \Pr\left[\,\mathcal{A}\Rightarrow 1 \text{ in } \bar{G}_0^*\,\right]| \leq \mathrm{Adv}^{\mathrm{init}}_{\mathcal{G},\mathsf{M},\gamma^*}(\bar{\mathcal{A}}_1,\mathcal{D})\ .$$

Moreover, notice that game G_0^* is identical to game Rob with challenge bit $b=1$. Putting this altogether, a standard argument implies that there exists adversaries $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ with the claimed query budgets such that

$$\mathrm{Adv}^{\mathrm{rob}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta}(\mathcal{A},\mathcal{D}) \leq 2\cdot \mathrm{Adv}^{\mathrm{init}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}_1,\mathcal{D}) + 2q_R \cdot (\mathrm{Adv}^{\mathrm{pres}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}_2) + 2q_R \cdot \mathrm{Adv}^{\mathrm{rec}}_{\mathcal{G},\mathsf{M},\beta,\gamma^*}(\mathcal{A},\mathcal{D}))\ ,$$

thereby completing the proof. ∎

E Proofs from Section 6

Proof of Lemma 1: Init security of HASH-DRBG.
Proof: We argue by a series of game hops, shown in Figure 12. We begin by defining game G_0, which is easily verified to be a rewriting of Init for HASH-DRBG and M^{H} with challenge bit $b=0$ in which we lazily sample the random oracle H. We additionally set a flag bad, although this does not affect the outcome of the game. It follows that

$$\Pr\left[\,G_0 \Rightarrow 1\,\right] = \Pr\left[\,\mathrm{Init}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*} \Rightarrow 1 \mid b=0\,\right]\ .$$

Next we define game G_1, which is identical to game G_0 except we change the way in which the random oracle H responds to queries. Namely, if H is queried more than once on one of the inputs $(i)_8 \,\|\, (L)_{32} \,\|\, I_1 \,\|\, N$ for $i \in [1,m]$ upon which it was queried to produce V_0^* (indicated in the

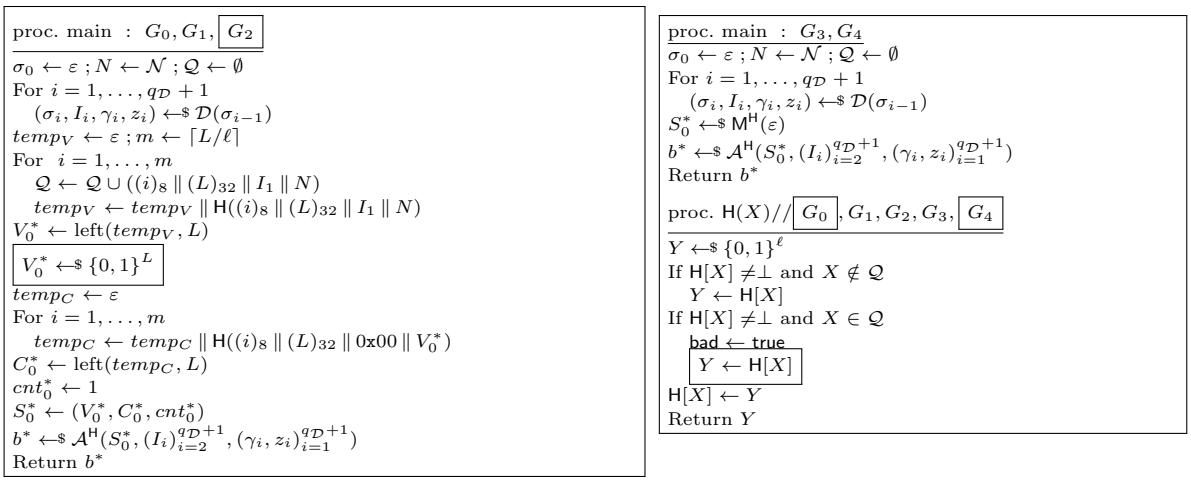

Fig. 12: Games for proof of Lemma 1 (Init security of HASH-DRBG).

pseudocode by the set $\mathcal{Q}$), then H responds with an independent random string rather than the value which was previously set. Notice that this event could either occur during the computation of C_0^* (if it happens that $\mathtt{0x00} \,\|\, V_0^* = I_1 \,\|\, N$), or due to a query made by $\mathcal{A}$ after receiving the challenge state. (Notice that due to the prepended counter, the queries made during the computation of V_0^* (resp. C_0^*) will never collide with eachother.) These games run identically unless the flag bad is set, and so the Fundamental Lemma of Game Playing implies that

$$|\Pr[\, G_0 \Rightarrow 1\,] - \Pr[\, G_1 \Rightarrow 1\,]| \leq \Pr[\, \mathsf{bad} = \mathsf{true} \text{ in } G_1\,] .$$

We let Coll denote the event that $\mathtt{0x00} \,\|\, V_0^* = I_1 \,\|\, N$. Notice that if Coll occurs, then bad will be set with probability one during the computation of C_0^*. Moreover, notice that if Coll does not occur then bad can only be set as the result of a query made by $\mathcal{A}$. It follows that

$$\begin{aligned}\Pr[\, \mathsf{bad} = \mathsf{true} \text{ in } G_1\,] &= \Pr[\, \mathsf{bad} = \mathsf{true} \wedge \mathsf{Coll} \text{ in } G_1\,] + \Pr[\, \mathsf{bad} = \mathsf{true} \wedge \neg\mathsf{Coll} \text{ in } G_1\,] \\ &\leq \Pr[\, \mathsf{Coll} \text{ in } G_1\,] + \Pr\left[\, \mathcal{A}^{\mathsf{H}}(S_0^*, (I_i)_{i=2}^{q_{\mathcal{D}}+1}, (\gamma_i, z_i)_{i=1}^{q_{\mathcal{D}}+1}) \text{ guesses } I_1 \text{ in } G_1 \,\right] .\end{aligned}$$

Before bounding this probability we first define game G_2, which is identical to G_1 except we overwrite the string V_0^* computed by querying H with an independent random string $V_0^* \leftarrow\$\ \{0,1\}^L$. In G_1, random oracle H responds to each query made to compute V_0^* with an independent random string (which is used at no other point in the game), and so it is straightforward to verify that these games are distributed identically. It follows that $\Pr[\, G_1 \Rightarrow 1\,] = \Pr[\, G_2 \Rightarrow 1\,]$, and so we may complete the upper bound of $\Pr[\, \mathsf{bad} = \mathsf{true} \text{ in } G_1\,]$ as follows:

$$\begin{aligned}&\Pr[\, \mathsf{Coll} \text{ in } G_1\,] + \Pr\left[\, \mathcal{A}(S_0^*, (I_i)_{i=2}^{q_{\mathcal{D}}+1}, (\gamma_i, z_i)_{i=1}^{q_{\mathcal{D}}+1}) \text{ guesses } I_1 \text{ in } G_1 \,\right] \\ &= \Pr[\, \mathsf{Coll} \text{ in } G_2\,] + \Pr\left[\, \mathcal{A}(S_0^*, (I_i)_{i=2}^{q_{\mathcal{D}}+1}, (\gamma_i, z_i)_{i=1}^{q_{\mathcal{D}}+1}) \text{ guesses } I_1 \text{ in } G_2 \,\right] \\ &\leq 2^{-L} + q_{\mathsf{H}} \cdot 2^{-\gamma^*} .\end{aligned}$$

All probabilities are over the coins of $\mathcal{A}$, $\mathcal{D}$, M and H. The first term in the inequality follows since $V_0^* \leftarrow\$\ \{0,1\}^L$, and so the probability that Coll occurs is upper bounded by 2^{-L}. The second term follows since in G_2 the challenge state S_0^* returned to $\mathcal{A}$ is entirely independent of the input I_1. As such, by the $(q_{\mathcal{D}}^+, \gamma^*)$-legitimacy of the sampler, the probability that $\mathcal{A}$ can guess I_1 with a single query conditioned on the given information is upper bounded by $2^{-\gamma^*}$. Taking a union bound over $\mathcal{A}$'s q_{H} queries then completes the argument.

Next we first define game G_3, which is the same as G_2 except we simply compute the challenge state S_0^* as $S_0^* \leftarrow\$\ \mathsf{M}^{\mathsf{H}}(\varepsilon)$. It is straightforward to verify that S_0^* is computed identically in both games, and that the redundant H queries from G_2 (made to compute V_0 which is subsequently overwritten, and which are removed in G_3) do not alter the outcome of the game. It follows that this is a syntactic change, and so $\Pr[\, G_2 \Rightarrow 1\,] = \Pr[\, G_3 \Rightarrow 1\,]$.

Next, we define game G_4, which is the same as G_3 except we return the random oracle to answer consistently on all points. However, since V_0^* is chosen independently at random in both G_3 and G_4 and so the values upon which the random oracle would 'lie' are not set (e.g., $\mathcal{Q} = \emptyset$) this does not alter the distribution of the game:

$$\Pr[G_3 \Rightarrow 1] = \Pr[G_4 \Rightarrow 1] .$$

Now, G_4 is identical to a rewriting of game Init with challenge bit $b = 1$ in which the random oracle is lazily sampled, and so

$$\Pr[G_4 \Rightarrow 1] = \Pr\left[\mathrm{Init}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*} \Rightarrow 1 \mid b = 1\right] .$$

Finally, putting this altogether, yields

$$\begin{aligned}\mathsf{Adv}^{\mathrm{init}}_{\mathcal{G},\mathsf{M},\gamma^*}(\mathcal{A},\mathcal{D}) &= |\Pr\left[\mathrm{Init}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*} \Rightarrow 1 \mid b = 0\right] - \Pr\left[\mathrm{Init}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*} \Rightarrow 1 \mid b = 1\right]| \\ &\leq |\Pr[G_0 \Rightarrow 1] - \Pr[G_4 \Rightarrow 1]| \\ &\leq q_{\mathsf{H}} \cdot 2^{-\gamma^*} + 2^{-L} .\end{aligned}$$

Next security of HASH-DRBG: a useful lemma. Consider the game $\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta}$ shown in Figure 13, in which we have assumed the PRNG $\mathcal{G}$ is not called with additional input. The advantage of an attacker $\mathcal{A}$ against a PRNG $\mathcal{G}$ with respect to masking function M^{H} is defined

$$\mathrm{Adv}^{\mathrm{nxt}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}) = |\Pr\left[\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 0\right] - \Pr\left[\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 1\right]| .$$

Looking ahead, proving the Next security of a PRNG $\mathcal{G}$ with respect to the masking function M will allow us to treat the proofs of preserving and recovering security in a more modular manner. We bound the Next security of HASH-DRBG with respect to the masking function M (Figure 4) in the following lemma. Looking ahead, this will allow us to treat the proofs of preserving and recovering security for HASH-DRBG in a more modular manner.

$\underline{\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta}}$
$\mathsf{H} \leftarrow\!\$\ \mathcal{H}\ ;\ b \leftarrow\!\$\ \{0,1\}$
$X \leftarrow\!\$\ \mathsf{salt}$
$S' \leftarrow\!\$\ \mathcal{A}^{\mathsf{H}}(X)\ ;\ S \leftarrow \mathsf{M}^{\mathsf{H}}(S')$
If $b = 0$
  $(R^*, S^*) \leftarrow\!\$\ \mathsf{next}^{\mathsf{H}}(X, S, \beta)$
Else $R^* \leftarrow\!\$\ \{0,1\}^{\beta}\ ;\ S^* \leftarrow\!\$\ \mathsf{M}^{\mathsf{H}}(S)$
$b^* \leftarrow\!\$\ \mathcal{A}^{\mathsf{H}}(X, R^*, S^*)$
Return $(b = b^*)$

Fig. 13: Game Next for a PRNG $\mathcal{G}$ and masking function M.

Lemma 9. *Let $\mathcal{G}$ = HASH-DRBG and masking function M^{H} be as specified in Theorem 2. Then for any adversary $\mathcal{A}$ in game Next against $\mathcal{G}$ making q_{H} queries to the random oracle H, it holds that*

$$\mathrm{Adv}^{\mathrm{nxt}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}) \leq \frac{q_{\mathsf{H}} \cdot n}{2^{\ell-1}} ,$$

where $n = \lceil \beta/\ell \rceil$ and $\mathsf{H} : \{0,1\}^{\leq\omega} \rightarrow \{0,1\}^{\ell}$.

Proof: We may assume without loss of generality that $\mathcal{A}$ never repeats a query to the random oracle H. Recall that $\mathsf{M}^{\mathsf{H}}(S')$ for $S' \in \mathcal{S}$ is distributed differently from $\mathsf{M}^{\mathsf{H}}(\varepsilon)$, and so there are two cases to consider depending on which type of state $\mathcal{A}$ outputs at the start of the challenge.

We begin by proving the case in which $\mathcal{A}$ outputs $S' \neq \varepsilon$. We argue by a series of game hops, shown on the left hand side of Figure 14. We begin by defining game G_0, which is a rewriting of game Next for HASH-DRBG and M^{H} with challenge bit $b = 0$ in which the random oracle is lazily sampled. In particular, notice how the procedure for computing masked states via application of M^{H} is included in the pseudocode at the appropriate point. We additionally set a flag bad, but this does not affect the outcome of the game. It follows that

$$\Pr[G_0 \Rightarrow 1] = \Pr\left[\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 0\right] .$$

Next, we define game G_1 which is identical to game G_0 except we change the way in which the random oracle H responds to queries. Namely, if H is queried on a point upon which it was already queried, it responds with an independent random string as opposed to the value previously set. These games run identically until the flag bad is set. It follows that

$$|\Pr[G_0 \Rightarrow 1] - \Pr[G_1 \Rightarrow 1]| \leq \Pr[\mathsf{bad} = 1 \text{ in } G_1] .$$

Next we define game G_2, which is identical to G_1 except we now overwrite the values of R^* and H_1 which were computed by querying H with independent random bit strings $R^* \leftarrow_{\$} \{0,1\}^{\beta}$ and $H_1 \leftarrow_{\$} \{0,1\}^{\ell}$. Since in G_1 the random oracle responds to each H query made to compute these variables with an independent random string which is used nowhere other than this in the game, it follows that these games are identically distributed, and so

$$\Pr[G_1 \Rightarrow 1] = \Pr[G_2 \Rightarrow 1] ; \text{ and}$$

$$\Pr[\mathsf{bad} = 1 \text{ in } G_1] = \Pr[\mathsf{bad} = 1 \text{ in } G_2] .$$

We now bound the probability of bad being set in G_2. During the computation of the challenge, H is queried on : **(1)** $data + j - 1 = V + j - 1$ for $j \in [1, n]$ during the computation of R^*; and **(2)** $0\mathtt{x}03 \,\|\, V$ upon which it was queried to generate the value H_1 (which is used to produce the final state variable V^*). Notice that since the former is shorter in length than the latter, no queries of the form **(1)** can collide with the query of form **(2)**. Moreover since $n < 2^L$, none of the queries of form **(1)** can collide with eachother. Since we have assumed that $\mathcal{A}$ never repeats a query, it follows that the only way that bad will be set is if an H query made by $\mathcal{A}$ in either the initial querying phase (i.e., before outputting S'), or after receipt of the challenge output / state pair, collides with one of the values queried by the challenger during the computation of the challenge. Moreover, notice that all queries which may cause bad to be set require $\mathcal{A}$ to guess an element in the set $\mathcal{V} = \{V, \ldots, V + n - 1\}$. We now bound the probability that $\mathcal{A}$ makes such a guess. Since knowledge of the challenge state S^* can only increase $\mathcal{A}$'s guessing probability, we may without loss of generality assume that $\mathcal{A}$ makes all of his q_{H} queries to H after receiving his challenge. It follows that

$$\begin{aligned}
\Pr[\mathsf{bad} = 1 \text{ in } G_2] &= \Pr\left[\mathcal{A}^{\mathsf{H}}(R^*, S^*) \text{ guesses a point in } \mathcal{V} \text{ in } G_2\right] \\
&\leq \sum_{j=1}^{n} \Pr\left[\mathcal{A}^{\mathsf{H}}(R^*, S^*) \text{ guesses } V + j - 1 \text{ in } G_2\right] \\
&= n \cdot \Pr\left[\mathcal{A}^{\mathsf{H}}(R^*, S^*) \text{ guesses } V \text{ in } G_2\right] .
\end{aligned}$$

Here the first inequality follows from taking a union bound over the n elements in $\mathcal{V}$. The following equality follows since guessing $V + j - 1 \bmod 2^L$ for some $j \in [1, n]$ is equivalent to guessing $V \bmod 2^L$.

In particular, notice that — while R^* is chosen independently at random in G_2 and so offers $\mathcal{A}$ no assistance in guessing the required state variables — $S^* = (V^*, C^*, cnt^*)$ *does* leak some information to $\mathcal{A}$ about the value of V. To see this, notice that V is computed as $V = V' + C' + cnt' + H_0$ where $H_0 \leftarrow_{\$} \{0,1\}^{\ell}$, $S' = (V', C', cnt')$ is the state output by $\mathcal{A}$ at the start of the game, and all addition is modulo 2^L. The challenge state component V^* is then computed as $V^* = V + C + cnt + H_1$ where $C = C'$ and $cnt = cnt + 1$, and so we may re-write V^* in the form

$$V^* = V' + 2 \cdot C' + (2 \cdot cnt' + 1) + (H_0 + H_1) ,$$

where recall all variables and sums are taken modulo 2^L. Now, all the variables on the right hand side of the above equality *except* H_0 and H_1 are known to $\mathcal{A}$; indeed, $\mathcal{A}$ selected V', C' and cnt' at the start of the game. As such, learning V^* reveals $(H_0 + H_1) \in [0, 2 \cdot (2^{\ell} - 1)]$ to $\mathcal{A}$. It is then straightforward to verify that guessing V given $R^*, S' = (V', C', cnt')$, and $S^* = (V^*, C^*, cnt^*)$ where recall $C^* = C'$ and $cnt^* = cnt + 2$ — that is to say, $\mathcal{A}$'s view of the experiment at this point — is equivalent to guessing H_0 given $(H_0 + H_1)$, where $H_0, H_1 \leftarrow_{\$} \{0,1\}^{\ell}$. More formally, we write e.g., $\mathbf{S}^*$ to denote the distribution of state S^*, and for brevity let $\alpha = V' + 2 \cdot C' + (2 \cdot cnt' + 1)$. It follows that for each $S' = (V', C', cnt')$ which may be output by $\mathcal{A}$, it holds that

$2^{-\tilde{\mathrm{H}}_{\infty}(\mathbf{V} | \mathbf{R}^*, \mathbf{S}^*, \mathbf{S}' = S')}$

$$
\begin{aligned}
&= \sum_{R^*,S^*} \max_V \Pr[\,\mathbf{V} = V \mid \mathbf{R}^* = R^*, \mathbf{S}^* = S^*, \mathbf{S}' = S'\,] \times \Pr[\,\mathbf{R}^* = R^*, \mathbf{S}^* = S^*, \mathbf{S}' = S'\,] \\
&= \sum_{R^*,S^*} \max_V \Pr[\,\mathbf{V} = V \wedge \mathbf{R}^* = R^*, \mathbf{S}^* = S^*, \mathbf{S}' = S'\,] \\
&= \sum_{\substack{S^*=(V^*,C',cnt'+2) \\ \text{for } V^* \in [\alpha, \alpha+2\cdot(2^\ell-1)]}} \max_V \Pr[\,(\mathbf{V} = V \text{ where } V = V' + C' + cnt' + H_0) \wedge (\mathbf{S}^* = S^* \text{ where } V^* = \alpha + (H_0 + H_1)\,] \\
&= \sum_{z \in [0, 2\cdot(2^\ell-1)]} \max_{H_0} \Pr[\,\mathbf{H_0} = H_0 \wedge (\mathbf{H_0} + \mathbf{H_1}) = z\,] \\
&= \sum_{z \in [0, 2\cdot(2^\ell-1)]} \max_{H_0} \Pr[\,(H_0 + \mathbf{H_1}) = z\,] \cdot \Pr[\,\mathbf{H_0} = H_0\,] \\
&= \sum_{z \in [0, 2\cdot(2^\ell-1)]} 2^{-\ell} \cdot 2^{-\ell} \\
&< 2^{-(\ell-1)}\ .
\end{aligned}
$$

All probabilities are over the random choice of $H_0, H_1 \leftarrow\!\!\$\ \{0,1\}^\ell$. The first equality follows from the definition of average-case min-entropy. The second equality follows from rearranging. The third equality follows by rewriting V and S^* in terms of $S' = (V', C', cnt')$ and noting that, since R^* is chosen randomly in G_2, it is independent of the distribution of state variables. The next equality follows since with S' fixed, the values of V and V^* are completely determined by $H_0 \in [1, 2^\ell - 1]$ and $(H_0 + H_1) \in [1, 2 \cdot (2^\ell - 1)]$. The following equality follows by rearranging. The last equality follows since for each $H_0 \in \{0,1\}^\ell$ and $z \in [0, 2\cdot(2^\ell - 1)]$, there is at most one $H_1 \in \{0,1\}^\ell$ such that $(H_0 + H_1) = z$. (Here we use the fact that $H_0, H_1 \in [0, 2^\ell - 1]$ and by assumption $L > \ell + 1$, and so no wraparound modulo 2^L occurs when computing $(H_0 + H_1)$.) This upper bounds the probability that $\mathcal{A}$ guesses V given a single guess, and so taking a union bound over $\mathcal{A}$'s q_{H} guesses, it follows that

$$\Pr[\,\mathsf{bad} \text{ in } G_2\,] \leq \frac{q_{\mathsf{H}} \cdot n}{2^{\ell-1}}\ .$$

Next we define G_3, which is identical to game G_2 except we compute S^* as $S^* = \mathsf{M}^{\mathsf{H}}(S)$ and omit the redundant random oracle queries which were made to compute the variables R^* and H_0 which are subsequently overwritten with random bit strings. It is straightforward to verify that these are syntactic changes, and so $\Pr[\,G_2 \Rightarrow 1\,] = \Pr[\,G_3 \Rightarrow 1\,]$.

Next we define game G_4, which is identical to G_3 except we return H to answer truthfully on all points; by an analogous argument to that used above these games run identically unless the flag bad is set. However, since the challenge output / state are generated randomly in both games and so no random oracle queries are made during the computation of the challenge state, it follows that bad will never be set and so this change does not alter the distribution of the game. Now G_4 is identical to Next with challenge bit $b = 1$ rewritten with a lazily sampled random oracle, and so it follows that

$$\Pr[\,G_4 \Rightarrow 1\,] = \Pr\left[\,\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 1\,\right]\ .$$

Putting this altogether we conclude that

$$
\begin{aligned}
\mathrm{Adv}^{\mathrm{nxt}}_{\mathsf{HASH\text{-}DRBG},\mathsf{M},\beta}(\mathcal{A}) &= \left|\Pr\left[\,\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 0\,\right] - \Pr\left[\,\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 1\,\right]\right| \\
&= \left|\Pr[\,G_0 \Rightarrow 1\,] - \Pr[\,G_4 \Rightarrow 1\,]\right| \\
&\leq \frac{q_{\mathsf{H}} \cdot n}{2^{\ell-1}}\ .
\end{aligned}
$$

We now consider the case in which $\mathcal{A}$ outputs state $S'_0 = \varepsilon$ at the start of game Next. In this case, the masking function chooses $V \leftarrow\!\!\$\ \{0,1\}^L$, sets $C = \mathsf{HASH\text{-}DRBG\_df}^{\mathsf{H}}(0\mathtt{x}00 \,\|\, V)$ and $cnt = 1$, and returns $S = (V, C, cnt)$. As we shall see, the success probability of an attacker in this case is less than that in the former case, and so the previous bound holds for both cases.

We argue by a series of game hops, shown on the right hand side of Figure 14. We begin by defining game G_0 which is a rewriting of game Next with challenge bit $b = 0$ for $\mathsf{HASH\text{-}DRBG}$

and M^{H} with a lazily sampled random oracle. Next we define game G_1, which is identical to game G_0 except we change H to respond to each query with an independent random string regardless of whether the value was previously set. These games run identically until the flag bad is set. An analogous argument to that used above invoking the Fundamental Lemma of Game Playing implies that

$$|\Pr[G_0 \Rightarrow 1] - \Pr[G_1 \Rightarrow 1]| \leq \Pr[\mathsf{bad} = 1 \text{ in } G_1] .$$

Next we define game G_2, which is identical to G_1 except we now overwrite the values of C, R^*, and H_1 which were computed by querying H, with independent random bit strings $C \leftarrow\!\!\$\ \{0,1\}^L$, $R^* \leftarrow\!\!\$\ \{0,1\}^\beta$, and $H_1 \leftarrow\!\!\$\ \{0,1\}^\ell$. As in the proof of the previous case, both games are identically distributed and so

$$\Pr[G_1 \Rightarrow 1] = \Pr[G_2 \Rightarrow 1] \text{ and ;}$$

$$\Pr[\mathsf{bad} = 1 \text{ in } G_1] = \Pr[\mathsf{bad} = 1 \text{ in } G_2] .$$

We now bound this latter probability. During the computation of the challenge, H is queried on: **(1)** $(i)_8 \,\|\, (L)_{32} \,\|\, \mathtt{0x00} \,\|\, V$ for $i \in [1, m]$ to compute C; **(2)** $V + j - 1$ for $j \in [1, n]$ to compute the output R^*; and **(3)** $\mathtt{0x03} \,\|\, V$ to compute H_1. Since the queries of form **(1)** cannot collide with eachother (due to the iterating counter) or with queries of forms **(2)** and **(3)** (since queries of type **(1)** are of a longer length than those of type **(2)** and **(3)**), an analogous argument to that made above implies that bad will only be set if $\mathcal{A}$ manages to guess one of the points queried by the challenger when computing the challenge. Moreover, all such queries require $\mathcal{A}$ to guess a point in the set $\mathcal{V} = \{V, \ldots, V + n - 1\}$. Since knowledge of the challenge state $S^* = (V^*, C^*, cnt^*)$ can only increase $\mathcal{A}$'s guessing probability, we may without loss of generality assume that $\mathcal{A}$ makes all of his q_{H} queries to H after receiving his challenge. An analogous argument to that above, taking a union bound, implies that

$$\Pr[\mathsf{bad} \text{ in } G_2] \leq n \cdot \Pr\left[\mathcal{A}^{\mathsf{H}}(R^*, S^*) \text{ guesses } V \text{ in } G_2\right] .$$

In particular, notice that possessing the challenge state leaks information to $\mathcal{A}$ about the value of the target V. Namely, $V^* = V + C + cnt + H_1$ where $H_1 \leftarrow\!\!\$\ \{0,1\}^\ell$, $C = C^*$, $cnt = 1$, and addition is modulo 2^L — notice how this differs from the previous case. As such, $\mathcal{A}$ knows that V corresponds to a random value in the interval $[V^* - C^* - 1 - (2^\ell - 1), V^* - C^* - 1]$. Given $\mathcal{A}$'s view of the experiment at this point, which consists of R^* amd $S^* = (V^*, C^*, cnt^*)$, it is straightforward to verify that guessing V is equivalent to guessing the value of $H_1 \leftarrow\!\!\$\ \{0,1\}^\ell$. As such, the probability that $\mathcal{A}$ guesses V with a single guess is equal to $2^{-\ell}$. Taking a union bound over $\mathcal{A}$'s q_{H} guesses then implies that $\mathcal{A}$'s probability of guessing V is upper bounded by $q_{\mathsf{H}} \cdot 2^{-\ell}$, and so

$$\Pr[\mathsf{bad} = 1 \text{ in } G_2] \leq \frac{q_{\mathsf{H}} \cdot n}{2^\ell} .$$

Next we define game G_3, which is identical to G_2 except we simply compute S^* as $S^* = \mathsf{M}^{\mathsf{H}}(S)$ and omit the redundant random oracle queries which were made to compute the variables R^* and H_1. As before, it is straightforward to verify that both games are identically distributed, and so $\Pr[G_2 \Rightarrow 1] = \Pr[G_3 \Rightarrow 1]$.

We then define game G_4, which is identical to G_3 except we no longer overwrite the string C with a random bit string. An analogous argument to that used above implies that $\Pr[G_3 \Rightarrow 1] = \Pr[G_4 \Rightarrow 1]$.

Next, we define game G_5, which is identical to G_4 except we return H to answer consistently on all points. By an analogous argument to that used above, G_4 and G_5 are identical until the flag bad is set. Since the only points queried to H during the computation of the challenge output / state in these games are during the computation of C, it follows that bad will only be set if $\mathcal{A}$ guesses one of the points queried to generate C. In turn, this may only occur if $\mathcal{A}$ guesses the point V. As such, an analogous argument to that above invoking the Fundamental Theorem of Game Playing and the fact that $V \leftarrow\!\!\$\ \{0,1\}^L$, implies that

$$\begin{aligned} |\Pr[G_4 \Rightarrow 1] - \Pr[G_5 \Rightarrow 1]| &\leq \Pr[\mathsf{bad} = 1 \text{ in } G_5] \\ &= \Pr[\mathsf{bad} = 1 \text{ in } G_3] \\ &\leq \frac{q_{\mathsf{H}}}{2^\ell} . \end{aligned}$$

Now G_5 is identical to Next with challenge bit $b = 1$ and rewritten with a lazily sampled random oracle, and so it follows that

$$\Pr\left[G_5 \Rightarrow 1\right] = \Pr\left[\,\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid\ b = 1\,\right] .$$

Putting this altogether, we conclude that

$$\begin{aligned}\mathrm{Adv}^{\mathrm{nxt}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}) &= \left|\Pr\left[\,\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid\ b = 0\,\right] - \Pr\left[\,\mathrm{Next}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid\ b = 1\,\right]\right| \\ &= \left|\Pr\left[G_0 \Rightarrow 1\right] - \Pr\left[G_5 \Rightarrow 1\right]\right| \\ &\leq \frac{q_{\mathsf{H}} \cdot (n+1)}{2^{\ell}} \leq \frac{q_{\mathsf{H}} \cdot n}{2^{\ell-1}} ,\end{aligned}$$

where the final inequality follows since $q_{\mathsf{H}} \geq 0$ and $n \geq 1$, thereby proving the bound in this case also.

Proof of Lemma 2: Pres security of HASH-DRBG.
Proof: Recall that in game Pres, the attacker $\mathcal{A}$ outputs a tuple $(S'_0, I_1, \ldots, I_d)$ at the start of his challenge which is then masked to give $S_0 \leftarrow\!\!\$\ \mathsf{M}^{\mathsf{H}}(S'_0)$. Our proof will proceed by arguing that after iteratively computing $S_i = \mathsf{refresh}^{\mathsf{H}}(S_{i-1}, I_i)$ for $i = 1, \ldots, d$, the resulting state S_d is indistinguishable from $\mathsf{M}^{\mathsf{H}}(\overline{S})$ for some state $\overline{S} \in \mathcal{S} \cup \{\varepsilon\}$ (which will be made explicit during the proof). This then allows us to reduce the Pres security of HASH-DRBG to the Next security of HASH-DRBG. Again since the state takes a different distribution after a reseed than after an output generation request, there are a number of cases to consider.

We may assume without loss of generality that $\mathcal{A}$ never repeats a query to H. We begin by considering the case in which the tuple $(S'_0, I_1, \ldots, I_d)$ output by $\mathcal{A}$ is such that: **(1)** $d \geq 1$ and so at least one reseed occurs during the challenge computation; and **(2)** $S'_0 \neq \varepsilon$. We argue by a series of game hops shown in Figure 15. We begin by defining game G_0, which is easily verified to be a rewriting of game Pres for HASH-DRBG and M^{H} with challenge bit $b = 0$ and with a lazily sampled random oracle. We have explicitly written out the code of the masking function at the point at which the state is masked at the start of the game. We additionally set a flag bad in G_0, but this does not affect the outcome of the game. It follows that

$$\Pr\left[G_0 \Rightarrow 1\right] = \Pr\left[\,\mathrm{Pres}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid\ b = 0\,\right] .$$

Next we define game G_1, which is identical to G_0 except that now if the random oracle H is queried more than once on any of the points $(i)_8 \,\|\, (L)_{32} \,\|\, \mathtt{0x01} \,\|\, V_{j-1} \,\|\, I_j$ for $i \in [1, m]$, $j \in [1, d]$ upon which it was queried during the iterative reseeds to compute $V_1, \ldots, V_d$ (indicated in the pseudocode by the set $\mathcal{Q}$), then H responds with an independent random string as opposed to the value previously set. These games run identically until the flag bad is set, and so the Fundamental Lemma of Game Playing implies that

$$\left|\Pr\left[G_0 \Rightarrow 1\right] - \Pr\left[G_1 \Rightarrow 1\right]\right| \leq \Pr\left[\mathsf{bad} = 1 \text{ in } G_1\right] .$$

Next we define game G_2, which is identical to G_1 except we overwrite each of the intermediate state variables V_j for $j \in [1, d]$ generated by querying H with an independent random bit string $V_j \leftarrow\!\!\$\ \{0,1\}^L$. Since in G_1, the strings returned in response to the H queries made to compute these variables are chosen independently at random and are used at no other point in the game, it follows that these games are identically distributed, and so

$$\Pr\left[G_1 \Rightarrow 1\right] = \Pr\left[G_2 \Rightarrow 1\right] ;$$

and

$$\Pr\left[\mathsf{bad} = 1 \text{ in } G_1\right] = \Pr\left[\mathsf{bad} = 1 \text{ in } G_2\right] .$$

We now bound this latter probability. Notice that bad will be set if any of the q_{H} H queries made by $\mathcal{A}$ collide with one of the points queried by the challenger when computing $V_1, \ldots, V_d$. The flag bad will also be set if there exist distinct $j, j' \in [0, d-1]$ such that $V_j \,\|\, I_{j+1} = V_{j'} \,\|\, I_{j'+1}$. (It is straightforward to verify that due to differing lengths and / or domain separation, none of the queries made to compute the constants C_j for $j \in [1, d]$, nor the queries made by $\mathsf{next}^{\mathsf{H}}$, will cause the flag bad to be set.)

Now, the challenge output / state pair are computed as $(R^*, S^*) \leftarrow \mathsf{next}^{\mathsf{H}}(S_d, \beta)$. Moreover notice that in G_2 the state S_d is computed as a function of the randomly chosen counter $V_d \leftarrow\!\!\$\ \{0,1\}^L$

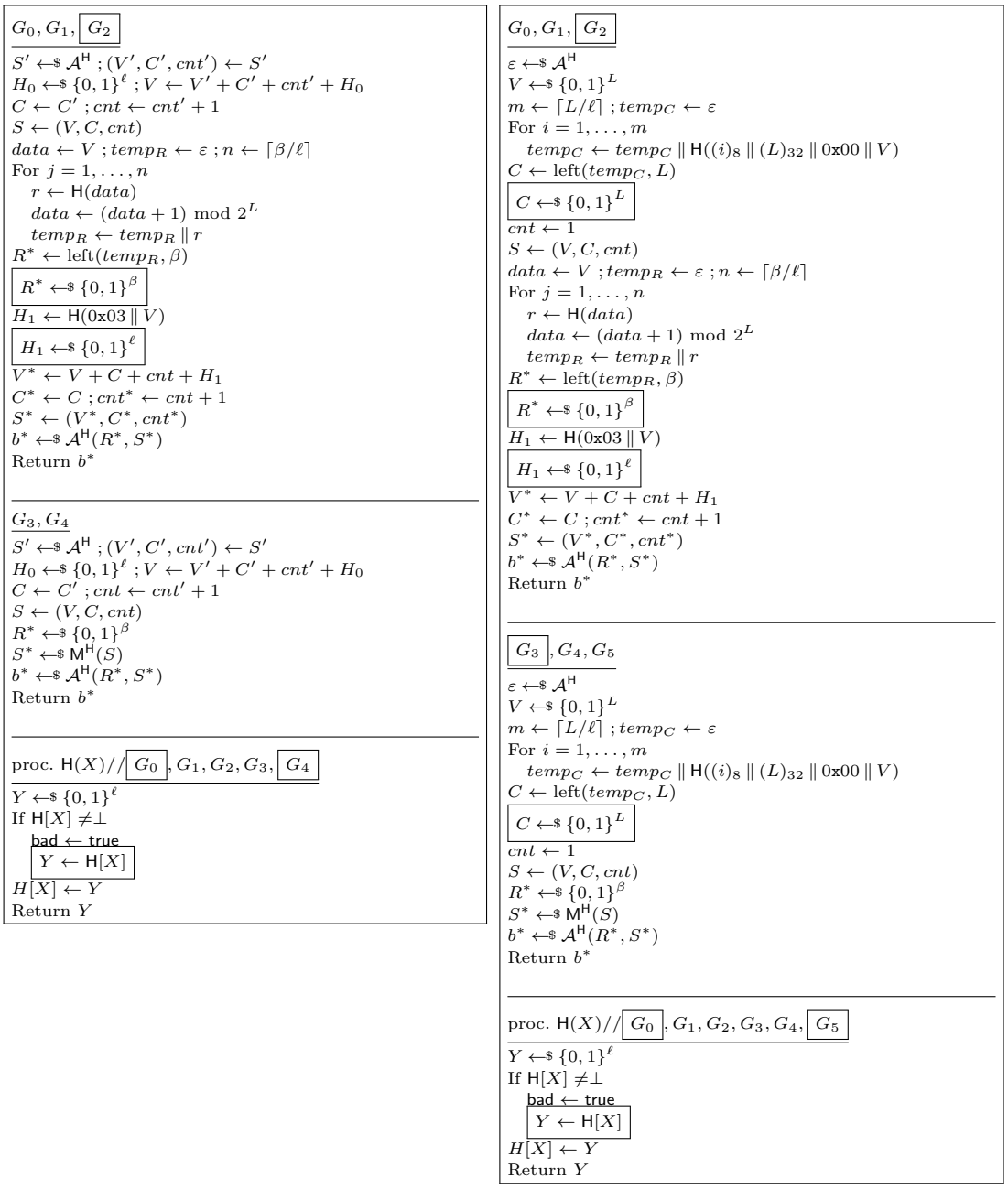

Fig. 14: Games for proof of Lemma 9 (Next security of HASH-DRBG). All addition is modulo 2^L.

as opposed to via iterative reseeding. As such, $\mathcal{A}$'s view of the experiment is independent of the intermediate state variables V_j, C_j for $j \in [0, d-1]$ and so we can without loss of generality imagine deferring the computation of these state variables until *after* $\mathcal{A}$ has finished making all of his queries to H. We now bound the probability that bad is set during this process.

We first note that bad will be set during the first reseed if $\mathcal{A}$ has made a query of the form $(i)_8 \,\|\, (L)_{32} \,\|\, 0\mathtt{x}01 \,\|\, V_0 \,\|\, I_1$ where $i \in [1, m]$. Any such query requires guessing V_0, and since V_0 is uniformly distributed over the set $[V'_0 + C'_0 + cnt'_0, V'_0 + C'_0 + cnt'_0 + 2^\ell - 1]$ it follows that the probability that $\mathcal{A}$ has made such a query is upper bounded by $\frac{q_\mathsf{H}}{2^\ell}$. Assuming that this event does not occur then bad will be set during the second reseed if either V_0 and V_1 collide, or $\mathcal{A}$ has already made a query of the correct form containing V_1. Since $V_1 \leftarrow\!\!\$\ \{0,1\}^L$ in G_2, it follows that bad is set during this reseed with probability $\frac{q_\mathsf{H}+1}{2^L}$ (the $q_\mathsf{H} \cdot 2^{-L}$ arising from $\mathcal{A}$'s queries and the additional 2^{-L} from the probability that $V_0 = V_1$ when $V_1 \leftarrow\!\!\$\ \{0,1\}^L$). Inductively applying the same argument yields that for all $j \in [2, d]$, the probability that bad is set during the j^{th} reseed is upper bounded by $\frac{q_\mathsf{H}+j-1}{2^L}$. Finally summing over these terms yields

$$\Pr[\,\mathsf{bad} = 1 \text{ in } G_2\,] \leq \frac{q_\mathsf{H}}{2^\ell} + \frac{(d-1)(2q_\mathsf{H}+d)}{2^{L+1}} \ .$$

Next we define G_3 in which we omit the iterative reseed calls and instead directly set $S_d \leftarrow\!\!\$\ \mathsf{M}^\mathsf{H}(\varepsilon)$. It is straightforward to verify that these games are identically distributed and so $\Pr[\,G_2 \Rightarrow 1\,] = \Pr[\,G_3 \Rightarrow 1\,]$. We then define game G_4, in which instead of computing R^* and S^* via $(R^*, S^*) \leftarrow \mathsf{next}^\mathsf{H}(S_d, \beta)$, we instead set $R^* \leftarrow\!\!\$\ \{0,1\}^\beta$ and $S^* \leftarrow \mathsf{M}^\mathsf{H}(S_d)$. We claim there exists an adversary $\mathcal{A}'$ in game Next against $\mathsf{HASH\text{-}DRBG}$ such that

$$|\Pr[\,G_3 \Rightarrow 1\,] - \Pr[\,G_4 \Rightarrow 1\,]| \leq \mathrm{Adv}^{\mathrm{nxt}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}') \leq \frac{q_\mathsf{H} \cdot n}{2^{\ell-1}} \ ,$$

and moreover $\mathcal{A}'$ makes q_H queries to his random oracle. To see this, notice that an attacker $\mathcal{A}'$ in game Next against $\mathsf{HASH\text{-}DRBG}$ and M^H can perfectly simulate $\mathcal{A}$'s view of the game as follows. For each of $\mathcal{A}$'s initial queries, $\mathcal{A}'$ simulates $\mathcal{A}$'s random oracle by forwarding all of $\mathcal{A}$'s queries to his own random oracle and returning the response. When $\mathcal{A}$ outputs a state S'_0, $\mathcal{A}'$ outputs the state ε as his challenge state receiving (R^*, S^*) in response. $\mathcal{A}'$ passes these to $\mathcal{A}$ and continues simulating $\mathcal{A}$'s random oracle by querying his own oracle as before. At the end of the game, $\mathcal{A}'$ outputs whatever bit $\mathcal{A}$ does. If $\mathcal{A}'$'s challenge bit is equal to 0 then this perfectly simulates game G_3, otherwise it perfectly simulates G_4, and so invoking Lemma 9 proves the claim. Moreover since $\mathcal{A}$ makes q_H queries, it follows that $\mathcal{A}'$ makes q_H queries also.

Finally in games G_5 and G_6 we reverse the earlier transitions to return to computing S_d via the process of iterative reseeding, and then in G_7 return the random oracle to respond consistently to all queries. By analogous arguments to those used above, $G_4 - G_6$ are identically distributed and G_6 and G_7 run identically unless bad is set, and so it follows that

$$|\Pr[\,G_4 \Rightarrow 1\,] - \Pr[\,G_7 \Rightarrow 1\,]| \leq \Pr[\,\mathsf{bad} = 1 \text{ in } G_5\,] \leq \frac{q_\mathsf{H}}{2^\ell} + \frac{(d-1)(2q_\mathsf{H}+d)}{2^{L+1}} \ .$$

Moreover, notice that G_7 is identical to Pres with challenge bit $b = 1$ and written with a lazily sampled random oracle, and so

$$\Pr[\,G_7 \Rightarrow 1\,] = \Pr\left[\,\mathrm{Pres}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 1\,\right] \ .$$

Putting this all together, we conclude that

$$\begin{aligned}
\mathrm{Adv}^{\mathrm{pres}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}) &\leq |\Pr\left[\,\mathrm{Pres}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 0\,\right] - \Pr\left[\,\mathrm{Pres}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 1\,\right]| \\
&\leq |\Pr[\,G_0 \Rightarrow 1\,] - \Pr[\,G_7 \Rightarrow 1\,]| \\
&\leq \frac{q_\mathsf{H} \cdot (n+1)}{2^{\ell-1}} + \frac{(d-1)(2q_\mathsf{H}+d)}{2^L} \ .
\end{aligned}$$

This proves the case in which $\mathcal{A}$ outputs $S'_0 \neq \varepsilon$ and $d \geq 1$. We now explain why this upper bound holds in all other cases too. Firstly for the case in which $\mathcal{A}$ outputs $S'_0 = \varepsilon$ at the start of his challenge and $d \geq 1$, the proof is identical except that when computing $S_0 \leftarrow\!\!\$\ \mathsf{M}^\mathsf{H}(S'_0)$, we set $V_0 \leftarrow\!\!\$\ \{0,1\}^L$ (and $C_0 \leftarrow \mathsf{HASH\text{-}DRBG\_df}^\mathsf{H}(0\mathtt{x}00 \,\|\, V_0)$, although this does not affect the proof). The increased entropy in the initial state can only make $\mathcal{A}$'s job harder and so the bound holds in this case also. Moreover (for both choices of initial state) if $d = 0$ and so no $\mathsf{refresh}$ calls are made, then in G_0 the challenge output / state are computed by applying next^H to the masked

state $S_0 \leftarrow\$ \mathsf{M}^{\mathsf{H}}(S'_0)$ where S'_0 is the state output by $\mathcal{A}$ at the start of his challenge. An analogous reduction to an attacker in game Next against HASH-DRBG and M^{H}, who passes S'_0 to his challenger and returns the response to $\mathcal{A}$, confirms the upper bound in this case also.

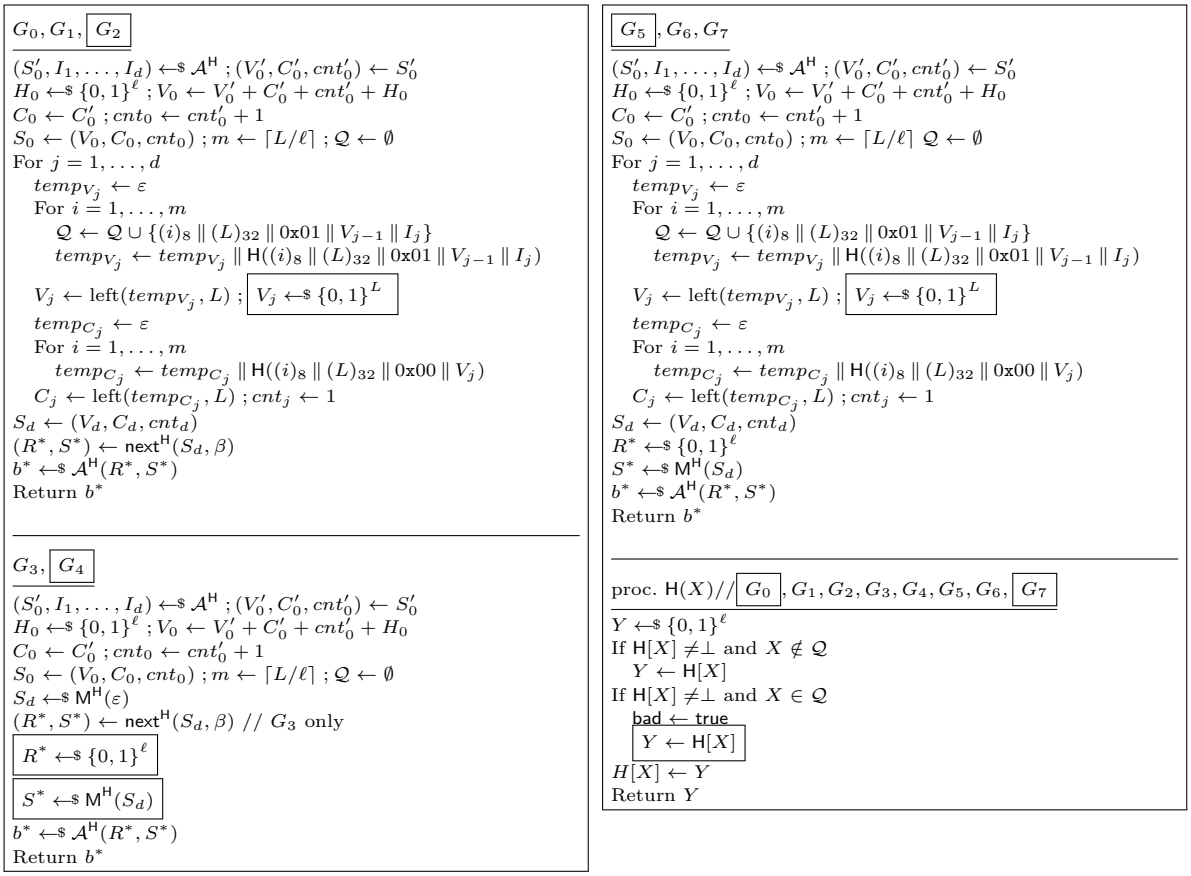

Fig. 15: Games for proof of Lemma 2 (Pres security of HASH-DRBG). All addition is modulo 2^L.

E.1 Recovering Security.

Analysis of HASH-DRBG extractor. Consider the game Ext shown in the left-hand panel of Figure 16, in which the advantage of an attacker and sampler pair $(\mathcal{A}, \mathcal{D})$ is defined

$$\mathrm{Adv}^{\mathrm{ext}}_{\mathcal{G},\mathsf{M},\gamma^*,q_{\mathcal{D}}}(\mathcal{A},\mathcal{D}) = 2 \cdot |\Pr\left[\,\mathrm{Ext}_{\mathcal{G},\mathsf{M},\gamma^*,q_{\mathcal{D}}}(\mathcal{A},\mathcal{D}) \Rightarrow 1\,\right] - \frac{1}{2}| \,.$$

In this game, the attacker $\mathcal{A}$ is challenged to distinguish a state of his choice which is iteratively reseeded with entropy samples with a collective entropy of at least γ^*-bits from a masked state $\mathsf{M}^{\mathsf{H}}(\varepsilon)$. In the following lemma we upper bound the success probability of an attacker in such a game against HASH-DRBG. We shall then use this result to bound the Rec security of HASH-DRBG.

Notation. We first introduce some notation. We let $\mathcal{J}$ denote the set of all queries made by $\mathcal{A}$ of the form

$$(i)_8 \,\|\, (L)_{32} \,\|\, 0\text{x}01 \,\|\, Y \,\|\, Z$$

where $i \in [1, m]$, $Y \in \{0,1\}^L$, and $Z \in \{0,1\}^{\geq 1}$ (e.g., points of the form which are queried to H to derive the V values during reseeds), and notice that all such strings can be decomposed unambiguously into these component parts. To each query $x \in \mathcal{J}$, we call $\overline{x} = (i, Y, Z)$ its associated decomposition; as we shall see, these shall be the components of these queries that are important for the proof.

Lemma 10. *Let $\mathcal{G} = \mathsf{HASH\text{-}DRBG}$ and masking function M^{H} be as specified in Theorem 2. Then for any adversary $\mathcal{A}$ in game Ext against $\mathsf{HASH\text{-}DRBG}$ making q'_{H} queries to the random oracle H of which q_J lie in $\mathcal{J}$, and any $(q_{\mathcal{D}}^{+}, \gamma^*)$-legitimate sampler $\mathcal{D}$, it holds that*

$$\mathrm{Adv}^{\mathrm{ext}}_{\mathcal{G},\mathsf{M},\gamma^*,q_{\mathcal{D}}}(\mathcal{A},\mathcal{D}) \leq \frac{q_J}{2^{\gamma^*}} + \frac{(d-1)\cdot(2q_J+d)+2q_J^2}{2^{L+1}} \ .$$

$\underline{\mathrm{Ext}_{\mathcal{G},\mathsf{M},\gamma^*,q_{\mathcal{D}}}(\mathcal{A},\mathcal{D})}$
$\mathsf{H} \leftarrow\$ \mathcal{H}\ ; b \leftarrow\$ \{0,1\}$
$\sigma_0 \leftarrow \varepsilon\ ; X \leftarrow\$ \mathrm{Seed}\ ; \mu \leftarrow 1$
For $k = 1, \ldots, q_{\mathcal{D}} + 1$
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow\$ \mathcal{D}(\sigma_{k-1})$
$(S_0, d) \leftarrow\$ \mathcal{A}^{\mathsf{H},\mathrm{Sam}}(X, (I_1, (\gamma_k, z_k))_{k=1}^{q_{\mathcal{D}}+1})$
If $\mu + d > (q_{\mathcal{D}} + 1)$ or $\sum_{i=\mu+1}^{\mu+d} \gamma_i < \gamma^*$
  Return $\perp$
$S_0 \leftarrow (V_0, C_0, cnt_0)$
If $b = 0$ then
  For $j = 1, \ldots, d$
    $S_j \leftarrow \mathsf{refresh}^{\mathsf{H}}(X, I_j, S_{j-1})$
Else $S_d \leftarrow\$ \mathsf{M}^{\mathsf{H}}(\varepsilon)$
$b^* \leftarrow\$ \mathcal{A}^{\mathsf{H}}(S_d, (I_k)_{k>\mu+d})$
Return $(b = b^*)$

$\underline{\mathrm{Sam}()}$
$\mu = \mu + 1$
Return I_μ

$\underline{\mathrm{Ext}_{\mathcal{G},\mathsf{M},\gamma^*,q_{\mathcal{D}}}(\mathcal{A},\mathcal{D})}$
$\mathsf{H} \leftarrow\$ \mathcal{H}\ ; b \leftarrow\$ \{0,1\}$
$\sigma_0 \leftarrow \varepsilon\ ; \mu \leftarrow 1$
For $k = 1, \ldots, q_{\mathcal{D}} + 1$
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow\$ \mathcal{D}(\sigma_{k-1})$
$(S_0, d) \leftarrow\$ \mathcal{A}^{\mathsf{H},\mathrm{Sam}}(X, (I_1, (\gamma_k, z_k))_{k=1}^{q_{\mathcal{D}}+1})$
If $\mu + d > (q_{\mathcal{D}} + 1)$ or $\sum_{i=\mu+1}^{\mu+d} \gamma_i < \gamma^*$
  Return $\perp$
$S_0 \leftarrow (V_0, C_0, cnt_0)$
If $b = 0$ then
For $j = 1, \ldots, d$
  $temp_{V_j} \leftarrow \varepsilon$
  For $i = 1, \ldots, m$
    $temp_{V_j} \leftarrow temp_{V_j} \,\|\, \mathsf{H}((i)_8 \,\|\, (L)_{32} \,\|\, \mathtt{0x01} \,\|\, V_{j-1} \,\|\, I_{\mu+j})$
  $V_j \leftarrow \mathrm{left}(temp_{V_j}, L)$
  $temp_{C_j} \leftarrow \varepsilon$
  For $i = 1, \ldots, m$
    $temp_{C_j} \leftarrow temp_{C_j} \,\|\, \mathsf{H}((i)_8 \,\|\, (L)_{32} \,\|\, \mathtt{0x00} \,\|\, V_j)$
  $C_j \leftarrow \mathrm{left}(temp_{C_j}, L)\ ; cnt_j \leftarrow 1$
$S_d \leftarrow (V_d, C_d, cnt_d)$
Else $S_d \leftarrow\$ \mathsf{M}^{\mathsf{H}}(\varepsilon)$
$b^* \leftarrow\$ \mathcal{A}^{\mathsf{H}}(S_d, (I_k)_{k>\mu+d})$
Return $(b = b^*)$

Fig. 16: Game Ext for a PRNG $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ (left) and pseudocode for proof of Lemma 10 (right).

Proof: Our proof uses the H-coefficient method as defined in Appendix A. We make the usual simplifying assumption that $\mathcal{A}$ is deterministic and that the sampler $\mathcal{D}$, having been initialized with coins $\omega \in \mathsf{coins}_{\mathcal{D}}$ where $\mathsf{coins}_{\mathcal{D}}$ denotes the coin space of the sampler, is deterministic also. Without loss of generality, we may assume that $\mathcal{A}$ makes precisely q_{H} queries, and never makes a redundant query. We begin by introducing some notation which will simplify our definition of *bad* transcripts. Within a transcript of an execution of game Ext, we let (x_ι, y_ι) denote that $\mathcal{A}$ queried x_ι to H and received y_ι in response. As with the sponge-extraction lemma of [20], we will modify game Ext so that $\mathcal{A}$ is given some additional information before outputting its challenge bit guess. However, these extra values are provided only after $\mathcal{A}$ has finished making all of its H queries, and so this extra information cannot influence $\mathcal{A}$'s choice of queries. Namely, we will additionally give to $\mathcal{A}$:

- The inputs $I_{\mu+1}, \ldots, I_{\mu+d}$ which $\mathcal{A}$ selected to use in the challenge computation; and
- The coins $\omega \in \mathsf{coins}_{\mathcal{D}}$ which the sampler $\mathcal{D}$ was initialized with.

Moreover, we shall further modify game Ext to change the way in which the random oracle H responds to queries. Namely, if H is queried on a point $x \in \mathcal{J}$ of the form $(*)_8 \,\|\, (L)_{32} \,\|\, \mathtt{0x01} \,\|\, Y \,\|\, Z$, the game computes $y^i = \mathsf{H}((i)_8 \,\|\, (L)_{32} \,\|\, \mathtt{0x01} \,\|\, Y \,\|\, Z)$ for *all* $i \in [1, m]$ and returns $y = y^1 \,\|\, \ldots \,\|\, y^m$ to $\mathcal{A}$. (Here we use $*$ to denote an arbitrary $i \in [1, m]$.) However, this is still counted as a single query made by $\mathcal{A}$. It is straightforward to see that such a change can only increase $\mathcal{A}$'s success probability since an attacker in the modified game can perfectly simulate the original game by returning block y^i of y to $\mathcal{A}$ in response to a query prefixed with index i. Our assumption that $\mathcal{A}$ never makes a redundant query now extends to him querying both $(i)_8 \,\|\, (L)_{32} \,\|\, \mathtt{0x01} \,\|\, Y \,\|\, Z$ and $(i')_8 \,\|\, (L)_{32} \,\|\, \mathtt{0x01} \,\|\, Y \,\|\, Z$ for $i, i' \in [1, m]$ and $i \neq i'$, since in the modified game the first query provides him with the answer to the latter. This modification shall simplify the subsequent proof, without increasing the attacker's success probability by too much for the parameter settings we are interested in. Indeed, in the worst case it gives the attacker m times as many queries, where for all parameter settings in the standard $m \leq 3$.

Recall that we write (x_ι, y_ι) to indicate that $\mathcal{A}$ queried x_ι to H and received y_ι in response. It is straightforward to verify from the pseudocode on the right hand side of Figure 16 that each execution of game Ext defines a transcript of the form

$$\tau = ((x_1, y_1), \ldots, (x_{q_\mathsf{H}}, y_{q_\mathsf{H}}), \omega, (V_0, C_0, cnt_0), (V_d, C_d, cnt_d), (I_k, \gamma_k, z_k)_{k=1}^{q_\mathcal{D}+1}, \mu, d) \ .$$

We say that a transcript τ is *compatible* if it may arise from an execution of Ext. We let T_0 denote the distribution of compatible transcripts produced from game Ext with challenge bit $b = 0$, and let T_1 denote the distribution of all such transcripts for Ext with challenge bit $b = 1$. With this in place, we define bad transcripts as follows:

Definition 2 (Bad transcripts for proof of Lemma 10). *Let τ be a compatible transcript with associated set of queries $(x_1, y_1), \ldots, (x_{q_\mathsf{H}}, y_{q_\mathsf{H}})$. Then τ is said to be* Bad *if among those queries there exists a subset of (not necessarily distinct) queries $\mathcal{X} = \{(u_1, v_1), \ldots, (u_d, v_d)\}$ such that:*

- *$u_\iota \in \mathcal{J}$ for all $(u_i, v_i) \in \mathcal{X}$; and*
- *For $j \in [1, d]$ it holds that u_j has decomposition $(*, X_{j-1}, I_{\mu+j})$ where $X_{j-1} = \mathrm{left}(v_{j-1}, L)$. (Here we define $\mathrm{left}(v_0, L) = V_0$ for notational brevity.)*

In words, a compatible transcript τ is Bad if it contains a set of queries which (with respect to our modification to the random oracle) are equivalent to those made by the challenger to compute the updated state component V_d. With this in place, our result is derived from the following two lemmas.

Lemma 11. *Let $\tau \in$ Good be a transcript. Then*

$$\frac{\Pr[\,\mathsf{T}_0 = \tau\,]}{\Pr[\,\mathsf{T}_1 = \tau\,]} \geq 1 - \frac{(d-1)\cdot(2q_J + d)}{2^{L+1}} \ ,$$

where recall that d denotes the index output by $\mathcal{A}$, and indicates the number of $\mathsf{refresh}^\mathsf{H}$ calls which are made during the computation of the challenge state.

Proof: We begin by noting that when $b = 1$ the challenge state S^* is computed as $\mathsf{M}^\mathsf{H}(\varepsilon)$ where recall that M^H chooses $V_d \leftarrow\!\!\$\ \{0,1\}^L$, sets $C_d = \mathsf{HASH\text{-}DRBG\_df}^\mathsf{H}(\mathrm{0x00} \,\|\, V_d)$ and $cnt_d = 1$, and returns $S_d = (V_d, C_d, cnt_d)$. Now for a random oracle H to be compatible with transcript τ, it must hold that:

- **(1)** $\mathsf{H}(x_\iota) = y_\iota$ for all $\iota \in [1, q_\mathsf{H}]$; and
- **(2)** H is consistent with $C_d = \mathsf{HASH\text{-}DRBG\_df}^\mathsf{H}(\mathrm{0x00} \,\|\, V_d)$.

It follows that

$$\begin{aligned}\Pr[\,\mathsf{T}_1 = \tau\,] =& \Pr[\,S_d = (V_d, \cdot, \cdot) \wedge \omega \text{ chosen} \wedge \mathsf{H} \text{ satisfies } \textbf{(1)} \text{ and } \textbf{(2)}\,] \\ &= 2^{-L} \cdot \Pr[\,\omega \text{ chosen} \wedge \mathsf{H} \text{ satisfies } \textbf{(1)} \text{ and } \textbf{(2)}\,] \ . \end{aligned} \tag{1}$$

Here the probability is over the choice of $\mathsf{H} \leftarrow\!\!\$\ \mathcal{H}$, $\omega \leftarrow\!\!\$\ \mathsf{coins}_\mathcal{D}$, and $V_d \leftarrow\!\!\$\ \{0,1\}^L$. (Recall that by assumption $\mathcal{A}$ is deterministic.) The final inequality follows since V_d is sampled independently at random in the ideal world, and so each V_d is selected with probability 2^{-L}.

We now consider the case that $\tau \in$ Good and $b = 0$. We write $S_d \leftarrow \mathsf{refresh}^\mathsf{H}(S_0, I_{\mu+1}, \ldots, I_{\mu+d})$ to denote the state returned by computing $S_i = \mathsf{refresh}^\mathsf{H}(S_{i-1}, I_{\mu+i})$ for $i = 1, \ldots, d$. Parsing $(V_d, C_d, cnt_d) \leftarrow S_d$, we let $q(\tau)$ denote the probability that $S_d \leftarrow \mathsf{refresh}^\mathsf{H}(S_0, I_{\mu+1}, \ldots, I_{\mu+d})$ yields the required counter V_d conditioned on ω being the chosen coins and H satisfying properties **(1)** and **(2)**. It follows that

$$\begin{aligned}\Pr[\,\mathsf{T}_0 = \tau\,] &= \Pr[\,S_d = (V_d, \cdot, \cdot) \wedge \omega \text{ chosen} \wedge \mathsf{H} \text{ satisfies } \textbf{(1)} \text{ and } \textbf{(2)}\,] \\ &= q(\tau) \times \Pr[\,\omega \text{ chosen} \wedge \mathsf{H} \text{ satisfies } \textbf{(1)} \text{ and } \textbf{(2)}\,] \\ &= q(\tau) \times 2^L \times \Pr[\,\mathsf{T}_1 = \tau\,], \end{aligned} \tag{2}$$

where the final equality follows from substituting in the term from line (1).

It remains to bound $q(\tau)$. One may imagine lazily sampling the random oracle H during the computation of V_d conditioned on it being consistent with the transcript via points **(1)** and **(2)**

as defined above. Let l' be maximal such that τ contains a set of queries $\{(u_1, v_1), \ldots, (u_{l'}, v_{l'})\}$ such that $u_j \in \mathcal{J}$ for each $j \in [1, l']$, and it holds that u_j has decomposition $(*, X_{j-1}, I_{\mu+j})$ where $X_{j-1} = \mathrm{left}(v_{j-1}, L)$. Since we have assumed that $\tau \in \mathsf{Good}$, it must be the case that $l' \in [0, d-1]$ (otherwise τ will be Bad).

For $j \in [l', d-1]$ we let $\neg\mathsf{Fresh}_j$ denote the event that either: **(a)** there exists a query $u_i \in \tau \cap \mathcal{J}$ such that u_i has decomposition $(*, V_j, Z)$ for some $Z \in \{0,1\}^{\geq 1}$; or **(b)** the state component V_j collides with a state component computed in a previous reseed call, $V_j \in \{V_{l'}, \ldots, V_{j-1}\}$. (Note that if V_j collides with a state variable in the set $\{V_0, \ldots, V_{l'-1}\}$, then this state will have formed part of an earlier query in the set $\{(u_1, v_1), \ldots, (u_{l'}, v_{l'})\} \subset \tau \cap \mathcal{J}$ and so is already accounted for by **(a)**.) Notice that if Fresh_j is true for some $j \in [l', d-1]$, then the queries made to compute the following state component V_{j+1} will all be on previously unqueried points. (We note that due to the prepended counter, none of the queries made to compute V_{j+1} can collide with eachother. Moreover, due to the separated domains, the queries made to compute the constants C_j for $j \in [1, d]$ can never collide with those made to compute the counters V_j.)

Now, the maximality of l' implies that the transcript τ contains no queries $u \in \mathcal{J}$ of the form $(*)_8 \,\|\, (L)_{32} \,\|\, 0\mathtt{x}01 \,\|\, V_{l'} \,\|\, Z$. As such, it follows that the $\Pr[\,\neg\mathsf{Fresh}_{l'}\,] = 0$. This implies that the state component $V_{l'+1}$ is computed as the result of all fresh H queries, and is therefore uniformly distributed over $\{0,1\}^L$. Now conditioned on $\mathsf{Fresh}_{l'}$, it follows that the probability that $\neg\mathsf{Fresh}_{l'+1}$ occurs is upper bounded by $\frac{(q_J+1)}{2^L}$. (Here, the $\frac{q_J}{2^L}$ accounts for the probability that there exists a query in $\tau \cap \mathcal{J}$ satisfying **(a)**, and the additional $\frac{1}{2^L}$ accounts for the probability that $V_{l'+1}$ collides with $V_{l'}$ as per **(b)**. Inductively applying this argument yields that for each $k \in [1, d-1-l']$, it holds that $\Pr\left[\,\neg\mathsf{Fresh}_{l'+k} \mid \wedge_{j=l'}^{l'+k-1} \mathsf{Fresh}_j\,\right] \leq \frac{(q_J+k)}{2^L}$. Using this bound, and the fact that $\Pr[\,\neg\mathsf{Fresh}_{l'}\,] = 0$ and $d-1-l' \leq d-1$, it follows that

$$\begin{aligned}
\Pr\left[\,\wedge_{j=l'}^{d-1}\mathsf{Fresh}_j\,\right] &\geq 1 - \sum_{k=0}^{d-1-l'} \Pr\left[\,\neg\mathsf{Fresh}_{l'+k} \mid \wedge_{j=l'}^{l'+k-1}\mathsf{Fresh}_j\,\right] \\
&\geq 1 - \sum_{k=1}^{d-1} \frac{(q_{\mathsf{H}}+k)}{2^L} \\
&= 1 - \frac{(d-1)\cdot(2q_J+d)}{2^{L+1}}\,.
\end{aligned}$$

Now, notice that if $\wedge_{j=l'}^{d-1}\mathsf{Fresh}_j$ is true then V_d is computed as a result of all fresh queries to H, and so the resulting state component is uniformly distributed over $\{0,1\}^L$. As such, the probability that the required value of V_d (as dictated by the transcript) is hit is equal to 2^{-L}. Putting this all together, we conclude that

$$\Pr[\,\mathsf{T}_0 = \tau\,] \geq \left(1 - \frac{(d-1)\cdot(2q_J+d)}{2^{L+1}}\right) \times \Pr[\,\mathsf{T}_1 = \tau\,]\,,$$

and so rearranging proves the lemma.

In the following lemma, we bound the probability that a compatible transcript in the ideal world is Bad.

Lemma 12. *Letting* Bad *denote the set of bad transcripts as defined above, it holds that*

$$\Pr[\,\mathsf{T}_1 \in \mathsf{Bad}\,] \leq \frac{q_J}{2^{\gamma^*}} + \frac{q_J^2}{2^L}\,.$$

Proof: Since we are now in the random world, the challenge state (V_d, C_d, cnt_d) is computed by choosing $V_d \leftarrow\!\!\$\ \{0,1\}^L$, setting $C_d = \mathsf{HASH\text{-}DRBG\_df}^{\mathsf{H}}(0\mathtt{x}00 \,\|\, V_d)$ and $cnt_d = 1$, and returning $S_d = (V_d, C_d, cnt_d)$. We let Chain denote the event that a compatible ideal world transcript τ contains a set of queries such that $\tau \in \mathsf{Bad}$. We define the notion of a *potential chain* as follows:

Definition 3. *We say that a sequence of (not necessarily distinct) queries* $\mathcal{X}' = (u_1, v_1), \ldots, (u_d, v_d)$ *constitutes a potential chain if*

- $(u_\iota, v_\iota) \in \mathcal{J}$ *for all* $(u_\iota, v_\iota) \in \mathcal{X}$*; and*

– *For $j \in [1,d]$ it holds that u_j has decomposition $(*, X_{j-1}, **)$ where $**$ denotes an arbitrary string in $\{0,1\}^{\geq 1}$, and $X_{j-1} = \mathrm{left}(v_{j-1}, L)$ where $\mathrm{left}(v_0, L) = V_0$.*

In words, a set of queries that form a potential chain are the same as those which form a bad transcript except we drop the condition that u_j contains the correct input $I_{\mu+j}$ in its decomposition. Moreover, notice that each potential chain defines a candidate sequence of inputs $Z = (Z_1, \ldots, Z_d)$, and a potential chain results in the transcript becoming Bad if $Z_j = I_{\mu+j}$ for $j \in [1,d]$.

We may visualize the potential chains in the form of an undirected graph as follows. We set $\omega_0 = V_0$ to be the root of the graph. We then use the queries in the transcript to construct paths of length d from the root by adding an edge between vertices ω_j, ω_k if there exists a query $(u,v) \in \tau \cap \mathcal{J}$, with decomposition $(*, \omega_j, **)$, and $\mathrm{left}(v, L) = \omega_k$. Notice that the potential chains correspond to paths of length d starting at the root V_0, and that since $\mathcal{A}$ gets q_{H} queries and $d \geq 1$, the graph can contain at most q_{H} edges. With this in place, we say that a transcript τ induces the event Coll if:

– There exists a query $(u_i, v_i) \in \tau \cap \mathcal{J}$ such $\mathrm{left}(v_i, L) = v_k$ for some query (u_k, v_k) where $1 \leq k < i$; or
– There exists a query $(u_i, v_i) \in \tau \cap \mathcal{J}$ such that $\mathrm{left}(v_i, L) = u_k$ for some query (u_k, v_k) where $1 \leq k \leq i$.

Now notice that if Coll does not occur then no newly added edge can loop back to a previously added vertex, and so the query graph must be a tree. In this case there can be at most one path to each leaf from the root. Since each node added to the graph corresponds to a query in $\tau \cap \mathcal{J}$ of which $\mathcal{A}$ makes q_J such queries, it follows that there can be at most q_J potential chains.

We now bound the probability that Coll occurs. Notice that conditioned on Coll having *not* occurred for the first $(k-1)$ queries in the set $\tau \cap \mathcal{J}$, then the k^{th} query of this form will be computed as the result of all fresh queries to H. As such, the resulting counter / vertex is uniformly distributed over $\{0,1\}^L$. It is then straightforward to verify that the probability that the k^{th} query sets Coll is upper bounded by $\frac{(2k-1)}{2^L}$. Summing over $k \in [1, q_J]$ then yields

$$\Pr[\,\mathsf{Coll}\,] \leq \sum_{k=1}^{q_J} \frac{(2k-1)}{2^L} = \frac{q_J^2}{2^L}\ .$$

Now as mentioned above, conditioned on Coll *not* occurring, a transcript can contain at most q_J potential chains. We now bound the probability that any of these q_J potential chain forms an actual chain. Since we are in the ideal world, $\mathcal{A}$'s view of game Ext is completely independent of the inputs $I_{\mu+1}, \ldots, I_{\mu+d}$ right up until the very end of the experiment when these values are revealed to $\mathcal{A}$ and which crucially is *after* $\mathcal{A}$ has finished making its queries to H. Therefore we can without loss of generality modify the experiment so that we only compute the states $S_i \leftarrow \mathsf{refresh}^{\mathsf{H}}(S_{i-1}, I_{\mu+i})$ for $i \in [1,d]$ *after* $\mathcal{A}$ has made all of his q_{H} queries (but before the unseen inputs $I_{\mu+1}, \ldots, I_{\mu+d}$ and sampler coins ω are revealed to $\mathcal{A}$ at the end of the game). We let τ' denote the transcript information available to $\mathcal{A}$ up to and including the point in game Ext at which he makes his last guess; namely

$$\tau' = ((x_1,y_1), \ldots, (x_q,y_q), (V_0, C_0, cnt_0), (V_d, C_d, cnt_d), \\ I_1, \ldots, I_\mu, I_{\mu+d+1}, \ldots, I_{q_{\mathcal{D}}}, (\gamma_k, z_k)_{k=1}^{q_{\mathcal{D}}+1}, \mu, d)\ .$$

Notice in particular that the inputs $I_{\mu+1}, \ldots, I_{\mu+d}$ which were used to compute $\mathcal{A}$'s challenge are *not* included, since they are still hidden from $\mathcal{A}$ at this stage. Moreover since V_d and C_d are independent of these entropy inputs in the ideal world, they reveal nothing to $\mathcal{A}$ about them. Now letting $\mathcal{F}$ denote the set of partial transcripts τ' for which Coll is false, it follows that

$$\begin{aligned} \Pr[\,\mathsf{Chain}\,] &\leq \Pr[\,\mathsf{Coll}\,] + \Pr[\,\mathsf{Chain} \wedge \neg\mathsf{Coll}\,] \\ &\leq \frac{q_J^2}{2^L} + \sum_{\tau' \in \mathcal{F}} \Pr[\,\mathsf{Chain} \mid \tau'\,] \cdot \Pr[\,\tau'\,]\ , \end{aligned} \tag{3}$$

where $\Pr[\,\tau'\,]$ denotes the probability that the execution Ext produces partial transcript τ' (that is to say: $\mathcal{A}$ makes the required set of queries, $V_d \leftarrow\!\!\$\ \{0,1\}^L$, and so on and so forth). Fix $\tau' \in \mathcal{F}$,

and let $Z_{\tau'}$ denote the set of potential chains within the partial transcript τ'. It follows that

$$\begin{aligned}
\Pr[\,\mathsf{Chain} \mid \tau'\,] &= \Pr[\,(I_{\mu+1},\ldots,I_{\mu+d}) \in Z_{\tau'} \mid \tau'\,] \\
&= \sum_{(Z_1,\ldots,Z_d)\in Z_{\tau'}} \Pr\left[\,\wedge_{i=\mu+1}^{\mu+d} Z_i = I_{\mu+i} \mid \tau'\,\right] \\
&\leq \sum_{(Z_1,\ldots,Z_d)\in Z_{\tau'}} \prod_{i=1}^{d} \Pr\left[\,Z_i = I_{\mu+i} \mid \wedge_{i=\mu+1}^{\mu+i-1} Z_i = I_{\mu+i}, \tau'\,\right] \\
&\leq \sum_{(Z_1,\ldots,Z_d)\in Z_{\tau'}} \prod_{i=1}^{d} 2^{-\gamma_{\mu+i}} \\
&\leq q_J \cdot 2^{-\gamma^*} \ .
\end{aligned}$$

Here the second to last inequality follows since the sampler is $(q_{\mathcal{D}}^+, \gamma^*)$-legitimate, since $\tau' \in \mathcal{F}$ implies that $Z_{\tau'}$ contains at most q_J potential chains. Substituting into equation (3) then proves the lemma.

Proof of Lemma 3: Rec security of HASH-DRBG.
Proof: We argue by a series of game hops, shown in Figure 17. We begin by defining game G_0, which is a rewriting of game Rec for HASH-DRBG with challenge bit $b = 0$. It follows that

$$\Pr[\,G_0 \Rightarrow 1\,] = \Pr\left[\,\mathrm{Rec}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta} \Rightarrow 1 \mid b = 0\,\right] \ .$$

Next we define game G_1 which is identical to G_0, except that rather than computing S_d via a sequence of $\mathsf{refresh}^{\mathsf{H}}$ calls we instead set $S \leftarrow\!\!\$\ \mathsf{M}^{\mathsf{H}}(\varepsilon)$. We claim that for any attacker / sampler pair $(\mathcal{A},\mathcal{D})$ making q_{H} queries to H, there exists a pair $(\mathcal{B},\mathcal{D})$ in game Ext against HASH-DRBG, where $\mathcal{B}$ makes $q_{\mathsf{H}} + n + 1$ queries to H of which at most q_{H} lie in $\mathcal{J}$, such that

$$|\Pr[\,G_0 \Rightarrow 1\,] - \Pr[\,G_1 \Rightarrow 1\,]| \leq \mathrm{Adv}^{\mathrm{ext}}_{\mathcal{G},\mathsf{M},\gamma^*}(\mathcal{B},\mathcal{D}) \ .$$

To see this, let $\mathcal{B}$ be the adversary who proceeds as follows. $\mathcal{B}$ simulates $\mathcal{A}$'s view of the game, forwarding $\mathcal{A}$'s Sam and H queries to his own oracles and returning the responses to $\mathcal{A}$. When $\mathcal{A}$ outputs a state / index pair (S_0, d), $\mathcal{B}$ forwards these to his challenger. $\mathcal{B}$ receives state S_d in response along with the remaining entropy samples. $\mathcal{B}$ computes $(R^*, S^*) \leftarrow \mathsf{next}^{\mathsf{H}}(S_d, \beta)$ using his own H oracle and returns these along with the entropy samples to $\mathcal{A}$, again using his oracle to answer all remaining queries. At the end of the game, $\mathcal{B}$ outputs whatever bit $\mathcal{A}$ does. Notice that if $\mathcal{B}$'s challenge bit is equal to 0 and so he receives the real state in his challenge then $\mathcal{B}$ perfectly simulates G_0; otherwise he perfectly simulates G_1. To verify the query budget, notice that $\mathcal{B}$ queries all of $\mathcal{A}$'s q_{H} queries to his own oracle and makes an additional $n + 1$ queries simulating $\mathsf{next}^{\mathsf{H}}$. Noting that none of the $n + 1$ queries made while simulating $\mathsf{next}^{\mathsf{H}}$ lie in $\mathcal{J}$ then proves the claim.

Next we define game G_2, which is identical to G_1 except we now set $R^* \leftarrow\!\!\$\ \{0,1\}^{\beta}$ and $S^* \leftarrow\!\!\$\ \mathsf{M}^{\mathsf{H}}(S_d)$ rather than computing these values as $(R^*, S^*) \leftarrow \mathsf{next}^{\mathsf{H}}(S_d, \beta)$. We claim that there exists an adversary $\mathcal{C}$ in game Next, who makes the same number of H queries as $\mathcal{A}$, such that

$$|\Pr[\,G_1 \Rightarrow 1\,] - \Pr[\,G_2 \Rightarrow 1\,] \leq \mathrm{Adv}^{\mathrm{nxt}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{C}) \leq \frac{q_{\mathsf{H}} \cdot n}{2^{\ell-1}} \ .$$

To see this, let $\mathcal{C}$ be the adversary who proceeds as follows. $\mathcal{C}$ uses the code of the sampler to generate all entropy samples, and passes the corresponding entropy estimates / side information to $\mathcal{A}$. For each of $\mathcal{A}$'s initial queries, $\mathcal{C}$ simulates $\mathcal{A}$'s random oracle by forwarding all of $\mathcal{A}$'s queries to his own random oracle and returning the response. $\mathcal{C}$ simulates $\mathcal{A}$'s Sam oracle by returning the appropriate entropy sample to $\mathcal{A}$. When $\mathcal{A}$ outputs a state S_0, $\mathcal{A}_2$ outputs the state ε as his challenge state, receiving (R^*, S^*) in response. $\mathcal{A}_2$ passes these to $\mathcal{A}$ along with the remaining entropy samples and continues simulating $\mathcal{A}$'s random oracle by querying his own oracle as before. At the end of the game, $\mathcal{C}$ outputs whatever bit $\mathcal{A}$ does. If $\mathcal{C}$'s challenge bit is equal to 0 then this perfectly simulates game G_1; otherwise he perfectly simulates G_2. As such, invoking Lemma 9 then proves the claim.

Next we define game G_3, which is identical to G_2 except we return to computing S_d via iterative reseeding as opposed to setting $S_d \leftarrow\!\!\$\ \mathsf{M}^{\mathsf{H}}(\varepsilon)$. An analogous argument to that above implies that

there exists an adversary $\mathcal{B}'$ making q_{H} queries to H (of which at most q_J lie in $\mathcal{J}$) such that

$$|\Pr[\,G_2 \Rightarrow 1\,] - \Pr[\,G_3 \Rightarrow 1\,]| \leq \mathrm{Adv}^{\mathrm{ext}}_{\mathcal{G},\mathsf{M},\gamma^*}(\mathcal{B}', \mathcal{D})\ ;$$

here the slightly lower query budget is because $\mathcal{B}'$ no longer needs to make the $n+1$ queries to H to simulate $\mathsf{next}^{\mathsf{H}}$. Now G_3 is identical to game Rec with challenge bit $b = 1$, and so

$$\Pr[\,G_3 \Rightarrow 1\,] = \Pr\left[\,\mathrm{Rec}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta} \Rightarrow 1 \mid\ b = 1\,\right]\ .$$

Putting this altogether, we conclude that

$$\begin{aligned}
\mathrm{Adv}^{\mathrm{rec}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta}(\mathcal{A},\mathcal{D}) &= |\Pr\left[\,\mathrm{Rec}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta} \Rightarrow 1 \mid\ b = 0\,\right] - \Pr\left[\,\mathrm{Rec}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta} \Rightarrow 1 \mid\ b = 1\,\right]| \\
&\leq |\Pr[\,G_0 \Rightarrow 1\,] - \Pr[\,G_3 \Rightarrow 1\,]| \\
&\leq \frac{q_{\mathsf{H}}}{2^{\gamma^*-1}} + \frac{(d-1)\cdot(2q_{\mathsf{H}}+d) + 2q_{\mathsf{H}}^2}{2^L} + \frac{q_{\mathsf{H}}\cdot n}{2^{\ell-1}}\,.
\end{aligned}$$

∎

$\underline{G_0}$
$\mathsf{H} \leftarrow\!\$\ \mathcal{H}\ ;\ \sigma \leftarrow \varepsilon\ ;\ \mu \leftarrow 1$
For $k = 1, \ldots, q_{\mathcal{D}} + 1$
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow\!\$\ \mathcal{D}(\sigma_{k-1})$
$(S_0, d) \leftarrow\!\$\ \mathcal{A}^{\mathsf{H},\mathrm{Sam}}((\gamma_i, z_i)_{i=1}^{q_{\mathcal{D}}+1})$
If $\mu + d > q_{\mathcal{D}} + 1$ or $\sum_{i=\mu+1}^{\mu+d} \gamma_i < \gamma^*$
  Return $\perp$
$(V_0, C_0, cnt_0) \leftarrow S_0$
For $j = 1, \ldots, d$
  $temp_{V_j} \leftarrow \varepsilon$
  For $i = 1, \ldots, m$
    $temp_{V_j} \leftarrow temp_{V_j} \,\|\, \mathsf{H}((i)_8 \,\|\, (L)_{32} \,\|\, \mathrm{0x01} \,\|\, V_{j-1} \,\|\, I_{\mu+j})$
  $V_j \leftarrow \mathrm{left}(temp_{V_j}, L)$
  $temp_{C_j} \leftarrow \varepsilon$
  For $i = 1, \ldots, m$
    $temp_{C_j} \leftarrow temp_{C_j} \,\|\, \mathsf{H}((i)_8 \,\|\, (L)_{32} \,\|\, \mathrm{0x00} \,\|\, V_j)$
  $C_j \leftarrow \mathrm{left}(temp_{C_j}, L)\ ;\ cnt_j \leftarrow 1$
$S_d \leftarrow (V_d, C_d, cnt_d)$
$(R^*, S^*) \leftarrow \mathsf{next}^{\mathsf{H}}(S_d, \beta)$
$b^* \leftarrow\!\$\ \mathcal{A}^{\mathsf{H}}(R^*, S^*, (I_k)_{k>\mu+d})$
Return b^*

$\underline{G_1}, \boxed{G_2}$
$\mathsf{H} \leftarrow\!\$\ \mathcal{H}\ ;\ \sigma \leftarrow \varepsilon\ ;\ \mu \leftarrow 1$
For $k = 1, \ldots, q_{\mathcal{D}} + 1$
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow\!\$\ \mathcal{D}(\sigma_{k-1})$
$(S_0, d) \leftarrow\!\$\ \mathcal{A}^{\mathsf{H},\mathrm{Sam}}((\gamma_i, z_i)_{i=1}^{q_{\mathcal{D}}+1})$
If $\mu + d > q_{\mathcal{D}} + 1$ or $\sum_{i=\mu+1}^{\mu+d} \gamma_i < \gamma^*$
  Return $\perp$
$S_d \leftarrow\!\$\ \mathsf{M}^{\mathsf{H}}(\varepsilon)$
$(R^*, S^*) \leftarrow \mathsf{next}^{\mathsf{H}}(S_d, \beta)$ // G_1 only
$\boxed{R^* \leftarrow\!\$\ \{0,1\}^{\ell}}$
$\boxed{S^* \leftarrow\!\$\ \mathsf{M}^{\mathsf{H}}(S_d)}$
$b^* \leftarrow\!\$\ \mathcal{A}^{\mathsf{H}}(R^*, S^*, (I_k)_{k>\mu+d})$
Return b^*

$\underline{G_3}$
$\mathsf{H} \leftarrow\!\$\ \mathcal{H}\ ;\ \sigma \leftarrow \varepsilon\ ;\ \mu \leftarrow 1$
For $k = 1, \ldots, q_{\mathcal{D}} + 1$
  $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow\!\$\ \mathcal{D}(\sigma_{k-1})$
$(S_0, d) \leftarrow\!\$\ \mathcal{A}^{\mathsf{H},\mathrm{Sam}}((\gamma_i, z_i)_{i=1}^{q_{\mathcal{D}}+1})$
If $\mu + d > q_{\mathcal{D}} + 1$ or $\sum_{i=\mu+1}^{\mu+d} \gamma_i < \gamma^*$
  Return $\perp$
$(V_0, C_0, cnt_0) \leftarrow S_0$
For $j = 1, \ldots, d$
  $temp_{V_j} \leftarrow \varepsilon$
  For $i = 1, \ldots, m$
    $temp_{V_j} \leftarrow temp_{V_j} \,\|\, \mathsf{H}((i)_8 \,\|\, (L)_{32} \,\|\, \mathrm{0x01} \,\|\, V_{j-1} \,\|\, I_{\mu+j})$
  $V_j \leftarrow \mathrm{left}(temp_{V_j}, L)$
  $temp_{C_j} \leftarrow \varepsilon$
  For $i = 1, \ldots, m$
    $temp_{C_j} \leftarrow temp_{C_j} \,\|\, \mathsf{H}((i)_8 \,\|\, (L)_{32} \,\|\, \mathrm{0x00} \,\|\, V_j)$
  $C_j \leftarrow \mathrm{left}(temp_{C_j}, L)\ ;\ cnt_j \leftarrow 1$
$S_d \leftarrow (V_d, C_d, cnt_d)$
$R^* \leftarrow\!\$\ \{0,1\}^{\ell}$
$S^* \leftarrow\!\$\ \mathsf{M}^{\mathsf{H}}(S_d)$
$b^* \leftarrow\!\$\ \mathcal{A}^{\mathsf{H}}(R^*, S^*, (I_k)_{k>\mu+d})$
Return b^*

$\underline{\mathrm{Sam}()}$
$\mu = \mu + 1$
Return I_μ

$\underline{\text{proc. } \mathsf{H}(X)//G_0, G_1, G_2, G_3}$
Return $\mathsf{H}(X)$

Fig. 17: Games for proof of Lemma 3 (Rec security of HASH-DRBG). All addition is modulo 2^L.

F Proofs from Section 7

Proof of Theorem 3.

Proof: Let $\mathcal{A}$ be the adversary who proceeds as follows. $\mathcal{A}$ makes a RoR query requesting ℓ-bits of output and for which $addin = \varepsilon$, receiving R^* is response. $\mathcal{A}$ then makes a Get query to receive state $S^* = (K^*, V^*, cnt^*)$. $\mathcal{A}$ then checks if

$$V^* \stackrel{?}{=} \mathsf{HMAC}(K^*, R^*) \ .$$

If so, $\mathcal{A}$ outputs 0; else he returns 1. It is straightforward to verify from the pseudocode description of next that if $b = 0$ and $\mathcal{A}$ receives a real output in his challenge then this relation will hold with probability one. Therefore, $\Pr\left[\ \mathcal{A} \Rightarrow 1 \text{ in } \mathrm{Fwd}^{\$,\mathcal{D},\mathcal{A}}_{\mathcal{G},\gamma^*} \ \mid\ b = 0\ \right] = 0$.

To bound $\Pr\left[\ \mathcal{A} \Rightarrow 1 \text{ in } \mathrm{Fwd}^{\$,\mathcal{D},\mathcal{A}}_{\mathcal{G},\gamma^*} \ \mid\ b = 1\ \right]$, we argue by a series of game hops. Let G_0 be identical to game $\mathrm{Fwd}^{\$}$ against $\mathcal{G}$ for $\mathcal{A}$ with challenge bit $b = 1$. Let $S_0 = (K_0, V_0, cnt_0)$ denote the initial state of $\mathcal{G}$, where recall that $K_0, V_0 \leftarrow\!\!\$\ \{0,1\}^\ell$. Notice that the RoR query made by $\mathcal{A}$ induces the challenger to update the state via $(R, S^*) \leftarrow \mathsf{next}(S_0, \ell)$. Since $b = 1$, $\mathcal{A}$ receives a random output $R^* \leftarrow\!\!\$\ \{0,1\}^\ell$; however, $\mathcal{A}$'s subsequent Get query allows him to learn the real state $S^* = (K^*, V^*, cnt^*)$. Now this state is computed as $V_0' \leftarrow \mathsf{HMAC}(K_0, V_0)$, $K^* \leftarrow \mathsf{HMAC}(K_0, V_0' || 0\mathtt{x}00)$, and $V^* \leftarrow \mathsf{HMAC}(K^*, V_0')$. We now define game G_1, which is identical to G_0 except we sample V_0', $K^* \leftarrow\!\!\$\ \{0,1\}^\ell$ instead of computing these variables via $\mathsf{HMAC}(K_0, \cdot)$. Since $K_0 \leftarrow\!\!\$\ \{0,1\}^\ell$, it is straightforward to verify that both G_0 and G_1 can be perfectly simulated by an attacker $\mathcal{B}_2$ in the PRF game against HMAC using two RoR queries and who runs in the same time as $\mathcal{A}$. This combined with the fact that, due to their disjoint domains, the queries made to compute V_0' and K^* can never collide, implies that

$$|\Pr[\ \mathcal{A} \Rightarrow 1 \text{ in } G_0\ ] - \Pr[\ \mathcal{A} \Rightarrow 1 \text{ in } G_1\ ] \leq \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{HMAC}}(\mathcal{B}_1, 2) \ .$$

Now recall that $\mathcal{A}$ outputs 1 if the relation $V^* = \mathsf{HMAC}(K^*, V_0') \stackrel{?}{=} \mathsf{HMAC}(K^*, R^*)$ does *not* hold. Since K^*, R^* and V_0' are all chosen randomly from $\{0,1\}^\ell$ in G_1, it follows that $\Pr[\ \mathcal{A} \Rightarrow 1 \text{ in } G_1\ ] = 1 - \epsilon_{coll}$ where

$$\epsilon_{coll} = \Pr\left[\ V^* = V' \ :\ V_0', R^*, K^* \leftarrow\!\!\$\ \{0,1\}^\ell\ ; V^* \leftarrow \mathsf{HMAC}(K^*, V_0'), V' \leftarrow \mathsf{HMAC}(K^*, R^*)\ \right] .$$

We claim that there exists an attacker $\mathcal{B}_2$ in the PRF security game against HMAC such that

$$\epsilon_{coll} \leq \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{HMAC}}(\mathcal{B}_2, 2) + 2^{-(\ell-1)} \ .$$

To see this, consider an attacker $\mathcal{B}_2$ in the PRF security game against HMAC who proceeds as follows. $\mathcal{B}_2$ chooses $V_0', R^* \leftarrow\!\!\$\ \{0,1\}^\ell$, queries these to his RoR oracle receiving V^*, V' in response, and outputs 1 if $V^* = V'$ and 0 otherwise. Notice that in the case that $\mathcal{B}_2$'s RoR oracle implements the real HMAC function then this perfectly simulates the experiment determining ϵ_{coll} and so $\Pr[\ \mathcal{B}_2 \Rightarrow 1 \ \mid\ b = 0\ ] = \epsilon_{coll}$. Moreover if $b = 1$ and so $\mathcal{B}_2$'s oracle implements a random function, it follows that

$$\begin{aligned}\Pr[\ \mathcal{B}_2 \Rightarrow 1 \ \mid\ b = 1\ ] &= \Pr[\ V^* = V' \wedge V_0' = R^*\ ] + \Pr[\ V^* = V' \wedge V_0' \neq R^*\ ] \\ &\leq 2^{-\ell} + 2^{-\ell} = 2^{-(\ell-1)} \ .\end{aligned}$$

To see this, notice that if $V_0' = R^*$ then $V^* = V'$ with probability one. Since $V_0', R^* \leftarrow\!\!\$\ \{0,1\}^\ell$, this implies the first term in the bound. Moreover, if $V_0' \neq R^*$, then V^*, V' are both the results of fresh queries to the random function and so are uniformly distributed over $\{0,1\}^\ell$, accounting for the second term in the bound. Combining the above via a standard argument then implies the claim.

Putting this altogether, and letting $\mathcal{B}$ be the attacker who tosses a coin to decide whether to run adversary $\mathcal{B}_1$ or $\mathcal{B}_2$, we conclude that

$$\begin{aligned}\mathrm{Adv}^{\text{fwd-\$}}_{\mathcal{G},\gamma^*}(\mathcal{A}, \mathcal{D}) &= |\Pr\left[\ \mathcal{A} \Rightarrow 1 \text{ in } \mathrm{Fwd}^{\$,\mathcal{D},\mathcal{A}}_{\mathcal{G},\gamma^*} \Rightarrow 1 \ \mid\ b = 0\ \right] - \Pr\left[\ \mathcal{A} \Rightarrow 1 \text{ in } \mathrm{Fwd}^{\$,\mathcal{D},\mathcal{A}}_{\mathcal{G},\gamma^*} \Rightarrow 1 \ \mid\ b = 1\ \right]| \\ &\geq 1 - 2 \cdot \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{HMAC}}(\mathcal{B}, 2) - 2^{-(\ell-1)} \ .\end{aligned}$$

∎

Proof of Lemma 4: Init security of HMAC-DRBG.
Proof: We argue by a series of game hops, shown in Figure 18. We assume without loss of generality that $\mathcal{A}$ never repeats a query to the random oracle HMAC. We begin by defining game G_0, which

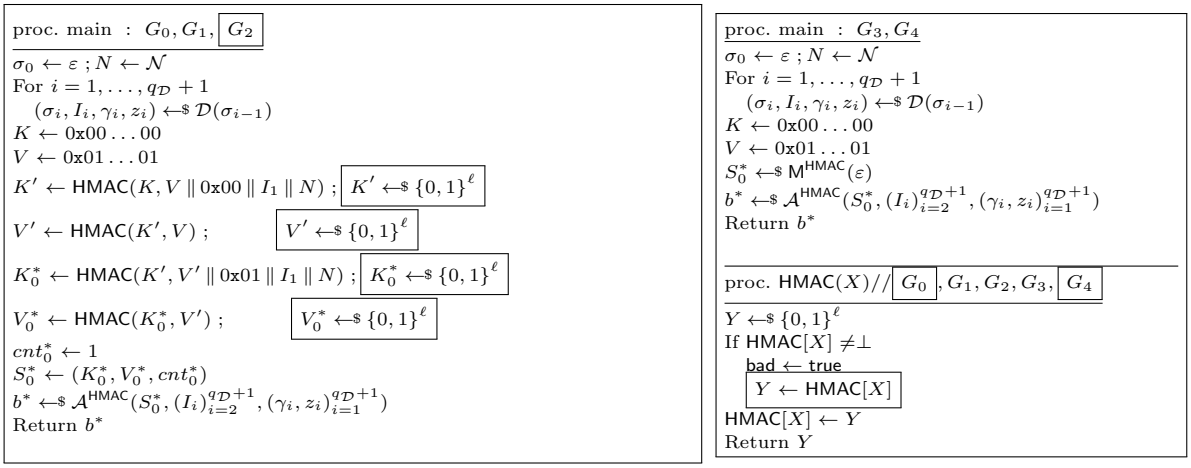

Fig. 18: Games for proof of Lemma 4 (Init security of HMAC-DRBG).

is a rewriting of game Init with $b = 0$ for HMAC-DRBG and $\mathsf{M}^{\mathsf{HMAC}}$ using a lazily sampled random oracle. We also set a flag bad, but this does not affect the outcome of the game. It holds that

$$\Pr[G_0 \Rightarrow 1] = \Pr\left[\mathrm{Init}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*} \Rightarrow 1 \mid b = 0\right] .$$

Next we define game G_1, which is identical to G_0 except we change the way in which the random oracle HMAC responds to queries. Namely if HMAC is queried on the same value more than once in G_1 then it responds with an independent random string as opposed to the value previously set. These games run identically until the flag bad is set, and so the Fundamental Lemma of Game Playing implies that

$$|\Pr[G_0 \Rightarrow 1] - \Pr[G_1 \Rightarrow 1]| \leq \Pr[\mathsf{bad} = 1 \text{ in } G_1] .$$

Next we define game G_2, which is identical to G_1 except during the challenge computation we overwrite each string returned in response to a query to HMAC with an independent random bit string drawn from $\{0,1\}^\ell$. Since in G_1 each string returned by the random oracle HMAC is chosen independently at random and used nowhere else in the game, these games are identically distributed:

$$\Pr[G_1 \Rightarrow 1] = \Pr[G_2 \Rightarrow 1]| \text{ and } \Pr[\mathsf{bad} = 1 \text{ in } G_1] = \Pr[\mathsf{bad} = 1 \text{ in } G_2] .$$

We now bound the probability that bad is set in G_2. Such an event may occur due to one of $\mathcal{A}$'s guesses, or due to a collision in the state variables during the challenge computation. We claim that

$$\Pr[\mathsf{bad} = 1 \text{ in } G_2] \leq q_{\mathsf{H}} \cdot ((1 + 2^{-2\ell}) \cdot 2^{-\gamma^*} + 2^{-(\ell-1)}) + 2^{-2\ell} .$$

To see this, let Guess denote the event that $\mathcal{A}$ guesses one of the points which was queried to HMAC by the challenger during the challenge computation. Let Coll denote the probability that bad is set during the challenge computation as the result of an accidental collision. A union bound implies that

$$\Pr[\mathsf{bad} = 1 \text{ in } G_2] = \Pr[(\mathsf{Guess} \vee \mathsf{Coll}) \text{ in } G_2] \leq \Pr[\mathsf{Guess} \text{ in } G_2] + \Pr[\mathsf{Coll} \text{ in } G_2] .$$

Now for Guess to occur, $\mathcal{A}$ must guess a point in the set

$$\{(K, V \| 0\text{x}00 \| I_1 \| N), (K', V), (K', V' \| 0\text{x}01 \| I_1 \| N), (K_0^*, V')\} .$$

Now, K, V, and K_0^* are all known to $\mathcal{A}$ (the former two being constant, and the latter being given to $\mathcal{A}$ as part of his challenge). However, all other variables are hidden from $\mathcal{A}$. Since $K', V' \leftarrow\$ \{0,1\}^\ell$, and the legitimacy of the sampler guarantees that I_1 has at least γ_1 bits of entropy conditioned on $\mathcal{A}$'s view of the experiment, a union bound over the elements in this set and $\mathcal{A}$'s q_{H} guesses implies that:

$$\Pr[\mathsf{Guess}] \leq q_{\mathsf{H}} \cdot ((1 + 2^{-2\ell}) \cdot 2^{-\gamma^*} + 2^{-(\ell-1)}) .$$

Moreover, due to the domain separation of the queries, it is straightforward to verify that the only way that Coll will occur is if $K_0^* = K'$ and $V' = V$ (in which case, the final query made by the challenger will not be fresh). Since $V', K_0^* \leftarrow\$ \{0,1\}^\ell$, it follows that this event occurs with probability at most $2^{-2\ell}$. Combining these observations then implies the claim.

Next we define G_3, which is identical to G_2 except we compute the challenge state directly as $S_0^* = \mathsf{M}^{\mathsf{HMAC}}(\varepsilon)$ and remove the now redundant queries to the random oracle HMAC. It is straightforward to verify that S_0^* is identically distributed in both games, and that this is a syntactic change. We then define game G_4, which is identical to G_3 except we return the random oracle HMAC to answer consistently if queried more than once on the same point. However, since no HMAC queries are made by the challenger in these games and we have assumed that $\mathcal{A}$ never repeats a query, HMAC will never be queried more than once on the same point, and so these games are identically distributed. It follows that $\Pr[G_2 \Rightarrow 1] = \Pr[G_3 \Rightarrow 1] = \Pr[G_4 \Rightarrow 1]$. Moreover, notice that G_4 is equivalent to game Init with challenge bit $b = 1$ and a lazily sampled random oracle, and so

$$\Pr[G_4 \Rightarrow 1] = \Pr\left[\mathrm{Init}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*} \Rightarrow 1 \mid b = 1\right] .$$

Putting this altogether, yields

$$\begin{aligned}\mathrm{Adv}^{\mathrm{init}}_{\mathcal{G},\mathsf{M},\gamma^*}(\mathcal{A},\mathcal{D}) &\le |\Pr[G_0 \Rightarrow 1] - \Pr[G_4 \Rightarrow 1]| \\ &\le q_{\mathsf{H}} \cdot ((1+2^{-2\ell}) \cdot 2^{-\gamma^*} + 2^{-(\ell-1)}) + 2^{-2\ell} .\end{aligned}$$

Proof of Lemma 5: Pres security of HMAC-DRBG.

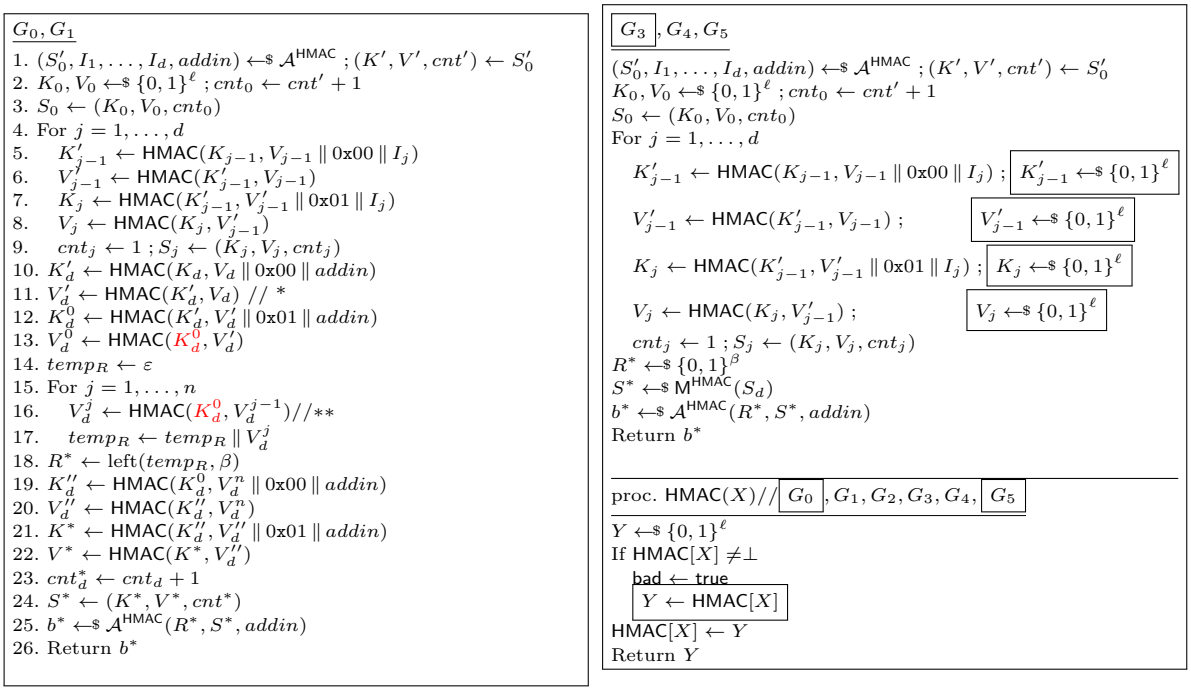

Fig. 19: Games for proof of Lemma 5 (Pres security of HMAC-DRBG).

Proof: We argue by a series of game hops, shown in Figure 19. We assume without loss of generality that $\mathcal{A}$ never repeats a query to the random oracle HMAC. We first define game G_0, which is a rewriting of game Pres with $b = 0$ for HMAC-DRBG and $\mathsf{M}^{\mathsf{HMAC}}$ using a lazily sampled random oracle. Recall that we assume $\mathcal{A}$ outputs $addin \neq \varepsilon$. We also set a flag bad, but this does not affect the outcome of the game. It holds that

$$\Pr[G_0 \Rightarrow 1] = \Pr\left[\mathrm{Pres}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 0\right] .$$

Next we define game G_1, which is identical to G_0 except we change the way in which the random oracle HMAC responds to queries. Namely in G_1 if HMAC is queried on the same value more than

once, it responds with an independent random string as opposed to the value previously set. These games run identically until the flag bad is set, and so the Fundamental Lemma of Game Playing implies that

$$|\Pr[G_0 \Rightarrow 1] - \Pr[G_1 \Rightarrow 1]| \leq \Pr[\mathsf{bad} = 1 \text{ in } G_1] .$$

Next, we define game G_2 (not shown), which is identical to G_1 except during the challenge computation, we overwrite each string returned in response to a query to HMAC with an independent random bit string drawn from $\{0,1\}^\ell$. Since in G_1 each string returned by the random oracle HMAC is chosen independently at random and used nowhere else in the game, these games are identically distributed, and so

$$\Pr[G_1 \Rightarrow 1] = \Pr[G_2 \Rightarrow 1]| \text{ and } \Pr[\mathsf{bad} = 1 \text{ in } G_1] = \Pr[\mathsf{bad} = 1 \text{ in } G_2] .$$

We now bound the probability that bad is set in G_2. Notice that bad may be set as the result of a query by $\mathcal{A}$ colliding with a query made by the challenger, or during the computation of the challenge output / state if two of the points queried to HMAC by the challenger collide.

Let Guess denote the event that one of $\mathcal{A}$'s queries collides with one of the points queried to HMAC by the challenger. Let Coll denote the probability that bad is set during the challenge computation as the result of an accidental collision. A union bound implies that

$$\Pr[\mathsf{bad} = 1 \text{ in } G_2] = \Pr[(\mathsf{Guess} \vee \mathsf{Coll}) \text{ in } G_2] \leq \Pr[\mathsf{Guess} \text{ in } G_2] + \Pr[\mathsf{Coll} \text{ in } G_2] .$$

One may verify from the pseudocode above that $4(d+2)+n$ HMAC computations are made during the challenge computation. (That is: four HMAC queries for each of the d refresh calls (lines 4 - 9); four queries for the 2 applications of update prior and post the production of output (lines 10 - 13 and 19 - 22 respectively); and $n = \lceil \beta/\ell \rceil$ queries to produce the output blocks (lines 15 - 17). Notice that each query made by $\mathcal{A}$ which would cause Guess to occur requires $\mathcal{A}$ to guess a key and counter pair. Notice that all such keys and counters are chosen uniformly from $\{0,1\}^\ell$ in G_2, and moreove these keys and counters are hidden from $\mathcal{A}$ *except* for: **(1)** the key K^* which forms part of the final HMAC query made by the challenger (line 22); and **(2)** the counters V_d^j for $j \in [1,n]$ (line 16) which are concatenated and truncated to form the output $R^* \leftarrow \text{left}(temp_R, \beta)$. All of these are revealed (partially in the case of V_d^n, if β is not a multiple of ℓ) to $\mathcal{A}$ as part of the challenge output / state pair. One may verify from the pseudocode that there are $n+2$ such partially known queries (e.g., the $n-1$ queries arising from the output generation loop; the queries in lines 19 and 20 for which the counter V_d^n is (at least partially) known, and the query on line 22 with known key K^*). The probability that $\mathcal{A}$ guesses a given unknown key / counter pair with a single guess is equal to $2^{-2\ell}$, while the probability that $\mathcal{A}$ guesses a given partially unknown pair is upper bounded by $2^{-\ell}$. Therefore taking a union bound over the $4(d+2)+n$ pairs queried, and $\mathcal{A}$'s q_{H} guesses, implies that:

$$\Pr[\mathsf{Guess} \text{ in } G_2] \leq q_{\mathsf{H}} \cdot ((4d+6) \cdot 2^{-2\ell} + (n+2) \cdot 2^{-\ell}) .$$

We now bound the probability that Coll occurs. We divide the queries made by the challenger into three classes as follows. For $K, V \in \{0,1\}^\ell$ and $I \in \{0,1\}^{\geq 1}$ we have: **(1)** queries of the form $(K, V \,\|\, 0\text{x}00 \,\|\, I)$; **(2)** queries of the form (K, V); and **(3)** queries of the form $(K, V \,\|\, 0\text{x}01 \,\|\, I)$. Notice that all queries made by the challenger during the challenge computation fall into one of these types. Moreover, notice that queries of different types can never collide due to their disjoint domains. Letting Coll_i for $i \in [1,3]$ denote that there is a collision amongst the type **(i)** queries, a union bound implies that

$$\Pr[\mathsf{Coll} \text{ in } G_2] \leq \Pr[\mathsf{Coll}_1 \text{ in } G_2] + \Pr[\mathsf{Coll}_2 \text{ in } G_2] + \Pr[\mathsf{Coll}_3 \text{ in } G_2] .$$

We first claim that $\Pr[\mathsf{Coll}_1 \text{ in } G_2] = \Pr[\mathsf{Coll}_3 \text{ in } G_2] = (d(d+3)+2) \cdot 2^{-(2\ell+1)}$. To see this, notice that the challenger makes a total of $(d+2)$ type **(1)** (resp. **(3)**) queries (d queries during the iterative reseeds, and a single query in the state update before and after output generation). Each of these queries consists of a *freshly sampled* key and counter chosen from $\{0,1\}^\ell$, by which we mean that the key (resp. the counter) will not be queried as part of any other query of that type barring an accidental collision. (Looking ahead, an example of queries which are *not* freshly sampled are those made to generate output blocks in line 17, for which the same key is used for each query.) It follows that the probability that any pair of queries collide is upper bounded by

$2^{-2\ell}$. Summing over the at most $\frac{1}{2}(d+1)(d+2)$ distinct pairs of queries and rearranging then proves the claim.

Next we claim that $\Pr[\,\mathsf{Coll}_2 \text{ in } G_2\,] \leq (d \cdot (2d+2n+7)+3n+6) \cdot 2^{-2\ell} + n(n+1) \cdot 2^{-\ell}$. To see this, first consider the set of type **(2)** queries made by the challenger up to the point in the pseudocode indicated by $*$ inclusive. There are $2d+1$ queries in this set, and each of these queries consists of a freshly sampled key and counter chosen randomly from $\{0,1\}^\ell$. Any given pair of such queries will collide with probability $2^{-2\ell}$; therefore summing over these pairs implies that the probability of a collision up to point $*$ is upper bounded by $d \cdot (2d+1) \cdot 2^{-2\ell}$. Now consider the set of type **(2)** queries made following $*$, and up to the point indicated by $**$ inclusive. While each counter for these queries is sampled uniformly from $\{0,1\}^\ell$, the key $K_d^0 \leftarrow\!\!\$\ \{0,1\}^\ell$ remains fixed across all queries (highlighted in red in the pseudocode). As such, each of the $n+1$ queries in this set collides with a previous query in the set with probability $2^{-\ell}$; summing over the $\frac{n(n+1)}{2}$ pairs of queries of this form (and rounding up by a factor of two for simplicity) contributes $n(n+1) \cdot 2^{-\ell}$ to the bound. Moreover, each such query collides with one of the $(2d+1)$ type **(2)** queries made up to point $*$ with probability $2^{-2\ell}$, contributing $(2d+1) \cdot (n+1) \cdot 2^{-2\ell}$ to the bound. Following this there are two more type **(2)** queries (lines 20 and 22), each consisting of a freshly sampled key and counter. Since there are $2d+n+2$ and $2d+n+3$ previous type **(2)** queries with which these points may collide, this adds a further $(4d+2n+5) \cdot 2^{-2\ell}$ to the bound. Summing over these terms and rearranging then proves the claim. Putting this altogether and simplifying the expression, we obtain:

$$\Pr[\,\mathsf{bad} = 1 \text{ in } G_2\,] \leq (q_{\mathsf{H}} \cdot (4d+6) + d \cdot (3d+2n+10) + 3n + 8) \cdot 2^{-2\ell} + (q_{\mathsf{H}} \cdot (n+2) + n(n+1)) \cdot 2^{-\ell}\,.$$

Next we first define game G_3, which is the same as G_2 except we simply compute the challenge / output state as $R^* \leftarrow\!\!\$\ \{0,1\}^\ell$ and $S^* \leftarrow\!\!\$\ \mathsf{M}^{\mathsf{HMAC}}(S_d)$ directly, and omit the now redundant HMAC queries which were previously made during their computation. It is straightforward to verify that S^* is computed identically in both games and that this is a syntactic change, and so

$$\Pr[\,G_2 \Rightarrow 1\,] = \Pr[\,G_3 \Rightarrow 1\,]\,.$$

Next we define game G_4, which is identical to G_3 except we no longer overwrite the responses to HMAC queries made by the challenger during the iterative reseeds with random bit strings. Since the random oracle HMAC responds to each query with an independent random string, regardless of whether that point has previously been queried, these games are identically distributed and so

$$\Pr[\,G_3 \Rightarrow 1\,] = \Pr[\,G_4 \Rightarrow 1\,]\,.$$

Finally we define game G_5, which is identical to G_4 except we revert the random oracle HMAC to answering consistently on all queries. These games run identically until the flag bad is set. In addition to being set as a result of the attacker's queries, in games $G_3 - G_5$ bad may also be set by the challenger during the iterative reseeds made to compute state S_d in the event of an accidental collision. As before, we have that

$$\begin{aligned}\Pr[\,\mathsf{bad} = 1 \text{ in } G_5\,] &= \Pr[\,\mathsf{bad} = 1 \text{ in } G_3\,] \\ &\leq \Pr[\,\mathsf{Guess} \text{ in } G_3\,] + \sum_{i=1}^{3} \Pr[\,\mathsf{Coll}_i \text{ in } G_3\,]\,.\end{aligned}$$

One may verify from the pseudocode that a total of $4d$ points are queried to HMAC during the iterative reseeds, each consisting of a uniformly random key and counter which are independent of $\mathcal{A}$'s view of the experiment. An analogous argument to that used previously, taking a union bound over $\mathcal{A}$'s q_{H} queries, then implies that $\Pr[\,\mathsf{Guess} \text{ in } G_3\,] \leq q_{\mathsf{H}} \cdot 4d \cdot 2^{-2\ell}$.

Moreover, there are d type **(1)** and d type **(3)** queries made during this process, each of which collides with probability at most $2^{-2\ell}$; it follows that $\Pr[\,\mathsf{Coll}_1\,] = \Pr[\,\mathsf{Coll}_3\,] \leq d \cdot (d-1) \cdot 2^{-(2\ell+1)}$. Finally, there are $2d$ type **(2)** queries made during this process, each of which collides with probability $2^{-2\ell}$. It follows that $\Pr[\,\mathsf{Coll}_2\,] \leq d \cdot (2d-1) \cdot 2^{-2\ell}$. Putting this altogether than

yields

$$|\Pr[G_4 \Rightarrow 1] - \Pr[G_5 \Rightarrow 1]| \leq \Pr[\mathsf{bad} = 1 \text{ in } G_3] \leq (q_{\mathsf{H}} \cdot 4d + d \cdot (3d-2)) \cdot 2^{-2\ell}$$

It is straightforward to verify that G_5 is equivalent to game Pres with $b = 1$ for HMAC-DRBG and $\mathsf{M}^{\mathsf{HMAC}}$ using a lazily sampled random oracle. It follows that

$$\Pr[G_0 \Rightarrow 1] = \Pr\left[\mathrm{Pres}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta} \Rightarrow 1 \mid b = 1\right] .$$

Putting this altogether and rearranging, we have

$$\begin{aligned}\mathrm{Adv}^{\mathrm{pres}}_{\mathcal{G},\mathsf{M},\beta}(\mathcal{A}) &\leq |\Pr[G_0 \Rightarrow 1] - \Pr[G_5 \Rightarrow 1]| \\ &\leq (q_{\mathsf{H}} \cdot (8d+6) + d \cdot (6d+2n+8) + 3n + 8) \cdot 2^{-2\ell} + (q_{\mathsf{H}} \cdot (n+2) + n(n+1)) \cdot 2^{-\ell} .\end{aligned}$$

Proof of Lemma 6: Rec security of HMAC-DRBG.

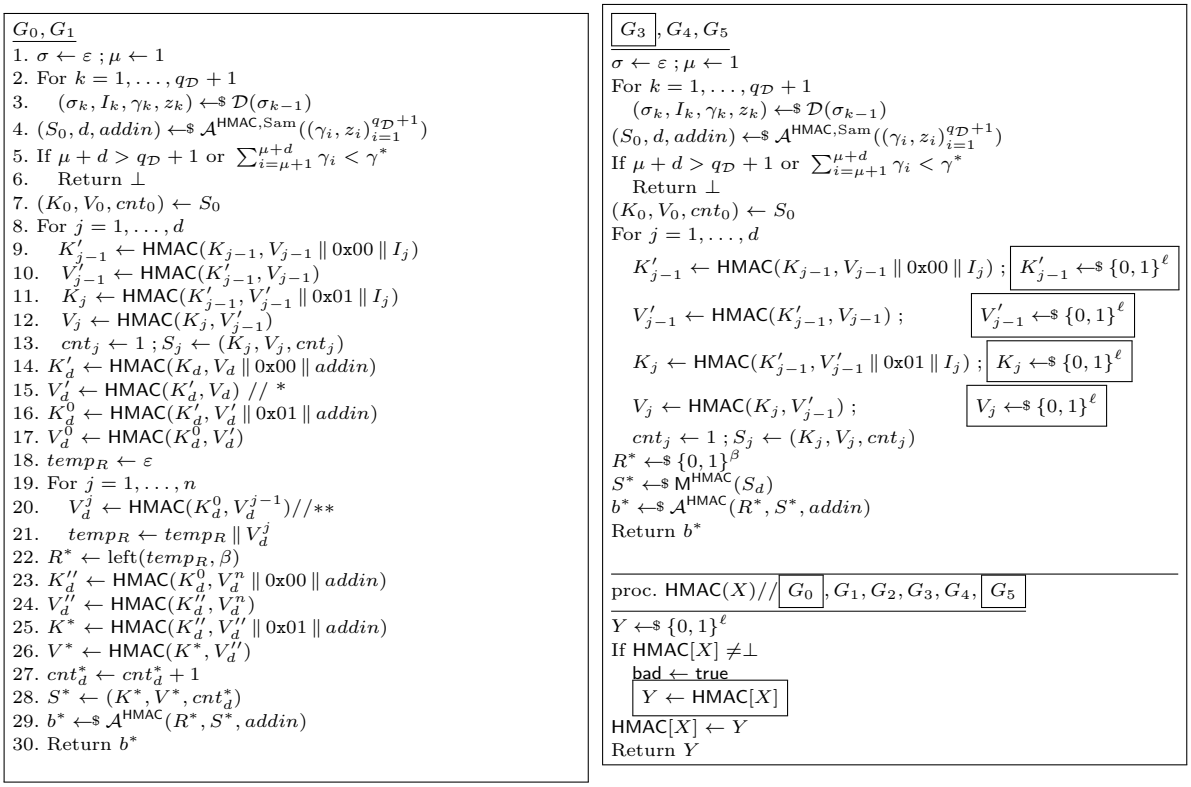

Fig. 20: Games for proof of Lemma 6 (Rec security of HMAC-DRBG).

Proof: We argue by a series of game hops, shown in Figure 20. The proof is similar to that of Lemma 5; we emphasize the differences here. We assume without loss of generality that $\mathcal{A}$ never repeats a query to the random oracle HMAC and require that $\mathcal{A}$ outputs $addin \neq \varepsilon$. We begin by defining game G_0, which is a rewriting of game Rec with $b = 0$ for HMAC-DRBG and $\mathsf{M}^{\mathsf{HMAC}}$ using a lazily sampled random oracle. We also set a flag bad, but this does not affect the outcome of the game. We have that

$$\Pr[G_0 \Rightarrow 1] = \Pr\left[\mathsf{Rec}^{\mathcal{A},\gamma^*}_{\mathcal{G},\mathsf{M},\gamma^*,\beta} \Rightarrow 1 \mid b = 0\right] .$$

Next we define games G_1 and G_2. In the former, we modify the random oracle HMAC to respond with an independent random string to all queries, regardless of whether the value has previously been set. In G_2 (not pictured), we overwrite each string returned in response to a random oracle

query made by the challenger with an independent random bit string. An analogous argument to that made in the proof of Lemma 5 implies that

$$|\Pr[G_0 \Rightarrow 1] - \Pr[G_2 \Rightarrow 1]| \leq \Pr[\mathsf{bad} = 1 \text{ in } G_2] .$$

We now bound the probability of this event. Defining the events Guess and Coll_i for $i \in [1,3]$ as before, a union bound implies that

$$\Pr[\mathsf{bad} = 1 \text{ in } G_2] \leq \Pr[\mathsf{Guess} \text{ in } G_2] + \sum_{i=1}^{3} \Pr[\mathsf{Coll}_i \text{ in } G_2] .$$

To bound the probability that Guess occurs, consider the set of queries made by the challenger, where we define queries of types **(1)** - **(3)** as in the proof of Lemma 5. We first note that each of the $2(d+2)+n$ type **(2)** queries made by the challenger consists of a random ℓ-bit key and ℓ-bit counter which are hidden from the attacker, with the exception of: **(a)** the first and final type **(2)** queries (lines 10 and 26 respectively); and **(b)** queries 2 to n of the $j = 1, \ldots, n$ loop (lines 19- 21), and the first type **(2)** query during the subsequent state update (line 24). (For the first query, the counter V_0 is chosen by the attacker. For the final query, the key K^* is given to the attacker as part of the challenge state. For all other queries, the counter is revealed to the attacker as part of the output R^*.) The probability that $\mathcal{A}$ guesses queries of the first type in a single guess is $2^{-2\ell}$; for the second type, this is upper bounded by $2^{-\ell}$. Putting this together and taking union bounds, the probability that $\mathcal{A}$ guesses one of the type **(2)** queries is upper bounded by $q_{\mathsf{H}} \cdot ((2d+2) \cdot 2^{-2\ell} + (n+2) \cdot 2^{-\ell})$.

Moreover, all of the d type **(3)** queries and all but the first of the d type **(1)** queries (line 9) made during the iterative reseeds in lines 8 - 13 requires $\mathcal{A}$ to guess a randomly chosen and hidden key and counter in addition to an entropy sample, which by the legitimacy of the sampler contains at least $2^{-\gamma^*}$ bits of entropy. It follows that the probability that $\mathcal{A}$ guesses any given one of these queries is upper bounded by $2^{-(\gamma^*+2\ell)}$. For the first type **(1)** query, the key and counter were chosen by $\mathcal{A}$; however, they must still guess the unknown entropy sample I_1 containing at least γ^* bits of entropy, and so the probability that $\mathcal{A}$ guesses this query is upper bounded by $2^{-\gamma^*}$. Finally, the remaining four type **(1)** and **(3)** queries (made as part of the state updates with *addin* before and after output generation (lines 14 - 17 and 24 - 26 respectively) each require $\mathcal{A}$ to guess a random and hidden key / counter pair, and so is guessed with probability at most $2^{-2\ell}$ — *except* the first type **(1)** query after output generation (line 23), for which the counter is (fully or partially) revealed in R^*, and so is guessed with probability at most $2^{-\ell}$. Taking union bounds, the probability that $\mathcal{A}$ guesses a type **(1)** or **(3)** query is upper bounded by $q_{\mathsf{H}} \cdot ((3 + (2d-1) \cdot 2^{-\gamma^*}) \cdot 2^{-2\ell} + 2^{-\gamma^*} + 2^{-\ell})$. Putting this altogether yields,

$$\Pr[\mathsf{Guess} \text{ in } G_2] \leq q_{\mathsf{H}} \cdot ((2d + 5 + (2d-1) \cdot 2^{-\gamma^*}) \cdot 2^{-2\ell} + 2^{-\gamma^*} + (n+3) \cdot 2^{-\ell}) .$$

Next we claim that $\Pr[\mathsf{Coll}_1 \text{ in } G_2]$ and $\Pr[\mathsf{Coll}_3 \text{ in } G_2]$ are bounded above by $(4d + 2 + d \cdot (d-1) \cdot 2^{-\gamma^*}) \cdot 2^{-(2\ell+1)}$. To see this, notice that a collision between a pair of the d type **(1)** (resp. type **(3)**) queries made during the iterative reseeds requires both the entropy input, and the key and counter (which for all but the first type **(1)** query are freshly sampled random bit strings), to collide. A union bound then implies that this probability is upper bounded by $d(d-1) \cdot 2^{-(\gamma^*+2\ell+1)}$. For any of the two type **(1)** (resp. type **(3)**) queries made during state updates before and after output generation, any collision with another query of the same type requires the randomly sampled key and counter to collide, an event which occurs with probability $d \cdot 2^{-2\ell}$ for the first such query, and $(d+1) \cdot 2^{-2\ell}$ for the second, contributing the additional $(2d+1) \cdot 2^{-2\ell}$ term to the bound. For the type **(2)** queries, it is straightforward to verify that the same argument as in the proof of Lemma 5 applies in this case also, and so $\Pr[\mathsf{Coll}_2] \leq (d \cdot (2d + 2n + 7) + 3n + 6) \cdot 2^{-2\ell} + n(n+1) \cdot 2^{-\ell}$. Putting this altogether and rearranging, yields

$$\Pr[\mathsf{bad} = 1 \text{ in } G_2] \leq q_{\mathsf{H}} \cdot ((2d+5+(2d-1) \cdot 2^{-\gamma^*}) \cdot 2^{-2\ell} + (d \cdot (2d+2n+11+(d-1) \cdot 2^{-\gamma^*}) + 3n + 8) \cdot 2^{-2\ell} + q_{\mathsf{H}} \cdot 2^{-\gamma^*} + (q_{\mathsf{H}} \cdot (n+3) + n(n+1)) \cdot 2^{-\ell} .$$

Next, we define games G_3, G_4, and G_5. Game G_3 is the same as G_2, except we compute the challenge / output state pair as $R^* \leftarrow\!\!\$\ \{0,1\}^\ell$ and $S^* \leftarrow\!\!\$\ \mathsf{M}(S_d)$, and omit the now redundant HMAC queries which were previously made during their computation. In G_4, we no longer overwrite the responses

to random oracle queries made by the challenger with random bit strings. Finally, in G_5 we return the random oracle HMAC to answer consistently on all queries. An analogous argument to that used in the proof of Lemma 5 implies that

$$|\Pr[G_2 \Rightarrow 1] - \Pr[G_5 \Rightarrow 1]| \leq \Pr[\mathsf{bad} = 1 \text{ in } G_3] .$$

We now bound this probability. As before, we have that

$$\begin{aligned}\Pr[\mathsf{bad} = 1 \text{ in } G_5] &= \Pr[\mathsf{bad} = 1 \text{ in } G_3] \\ &\leq \Pr[\mathsf{Guess} \text{ in } G_3] + \sum_{i=1}^{3} \Pr[\mathsf{Coll}_i \text{ in } G_3] .\end{aligned}$$

Firstly, notice that there are $2d$ type **(2)** queries made by the challenger in G_3. Each of these consists of a freshly sampled random key and counter, except for the first query where the counter is known to $\mathcal{A}$. An analogous argument to that used previously implies that the probability that $\mathcal{A}$ guesses one of these points is upper bounded by $q_{\mathsf{H}} \cdot ((2d-1) \cdot 2^{-2\ell} + 2^{-\ell})$. Each of the d type **(1)** (resp. type **(3)**) queries consists of a random key / counter and entropy sample, except for the first type **(1)** query for which the key and counter are known to $\mathcal{A}$. A union bound then implies that the probability that $\mathcal{A}$ guesses one of the type **(1)** or **(3)** queries is upper bounded by $q_{\mathsf{H}} \cdot ((2d-1) \cdot 2^{-\gamma^*} \cdot 2^{-2\ell} + 2^{-\gamma^*})$. Putting this altogether, implies that:

$$\Pr[\mathsf{Guess} \text{ in } G_3] \leq q_{\mathsf{H}} \cdot ((2d - 1 + (2d-1) \cdot 2^{-\gamma^*}) \cdot 2^{-2\ell} + 2^{-\gamma^*} + 2^{-\ell}) .$$

We now bound the collision probabilities. There are d type **(1)** and d type **(3)** queries made in G_3, each of which collides with another query of its type with probability at most $2^{-(2\ell+\gamma^*)}$; it follows that $\Pr[\mathsf{Coll}_1 \text{ in } G_3] = \Pr[\mathsf{Coll}_3 \text{ in } G_3] \leq d \cdot (d-1) \cdot 2^{-\gamma^*} \cdot 2^{-(2\ell+1)}$. Finally, each of the $2d$ type **(2)** queries in G_3 collides with probability $2^{-2\ell}$; taking a union bound, it follows that $\Pr[\mathsf{Coll}_2 \text{ in } G_3] \leq d \cdot (2d-1) \cdot 2^{-2\ell}$. Putting this all together then yields

$$\begin{aligned}\Pr[\mathsf{bad} = 1 \text{ in } G_3] \leq q_{\mathsf{H}} \cdot (2d - 1 + (2d-1) \cdot 2^{-\gamma^*}) \cdot 2^{-2\ell} + (d \cdot (2d - 1 + (d-1) \cdot 2^{-\gamma^*})) \cdot 2^{-2\ell} \\ + q_{\mathsf{H}} \cdot 2^{-\gamma^*} + q_{\mathsf{H}} \cdot 2^{-\ell} .\end{aligned}$$

Moreover, it is straightforward to verify that G_5 is equivalent game Rec with $b = 1$ for $\mathsf{HMAC\text{-}DRBG}$ and $\mathsf{M}^{\mathsf{HMAC}}$ with a lazily sampled random oracle. It follows that

$$\Pr[G_5 \Rightarrow 1] = \Pr\left[\mathsf{Rec}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta} \Rightarrow 1 \mid b = 1\right] .$$

Putting this altogether, we have

$$\begin{aligned}&\mathrm{Adv}^{\mathrm{rec}}_{\mathcal{G},\mathsf{M},\gamma^*,\beta}(\mathcal{A},\mathcal{D}) \leq |\Pr[G_0 \Rightarrow 1] - \Pr[G_5 \Rightarrow 1]| \\ &\leq q_{\mathsf{H}} \cdot (2d + 2 + (2d-1) \cdot 2^{-\gamma^*}) \cdot 2^{-(2\ell-1)} + (d \cdot (4d + 2n + 10 + (d-1) \cdot 2^{-(\gamma^*-1)}) + 3n + 8) \cdot 2^{-2\ell} \\ &\quad + (q_{\mathsf{H}} \cdot (n+4) + n(n+1)) \cdot 2^{-\ell} + q_{\mathsf{H}} \cdot 2^{-(\gamma^*-1)} .\end{aligned}$$

G Additional Algorithms and Sample Parameters

In this section, we describe the setup and derivation function algorithms for $\mathsf{CTR\text{-}DRBG}$ which were omitted from Section 3, and give examples of typical parameter settings.

G.1 Additional Algorithms

The setup algorithm shown here is for $\mathsf{CTR\text{-}DRBG}$ implemented with a derivation function; the alternative algorithm is very similar. The derivation function $\mathsf{CTR\text{-}DRBG\_df}$ returns an error if $num\_bits > 512$; we omit this check from the pseudocode below for simplicity. Note the order in which the output blocks are parsed by the subroutine BCC; this is the step in which the description of the algorithm from [37] differs from the standard[11].

[11] This step is written in the standard as "Starting with the leftmost bits of data, split *data* into n blocks of ℓ bits each, forming $block_1$ to $block_n$".

setup
Require: I, $nonce$
Ensure: $S_0 = (K_0, V_0, cnt_0)$
$seed\_material \leftarrow I \,\|\, nonce$
$seed\_material \leftarrow \mathsf{CTR\text{-}DRBG\_df}(seed\_material, (\kappa + \ell))$
$K \leftarrow 0^\kappa$; $V \leftarrow 0^\kappa$
$(K_0, V_0) \leftarrow \mathsf{update}(seed\_material, K, V)$
$cnt_0 \leftarrow 1$
Return (K_0, V_0, cnt_0)

BCC
Require: $K, data$
Ensure: $output\_block$
$chain \leftarrow 0^\ell$
$n \leftarrow len(data)/\ell$
$block_1 \,\|\, \ldots \,\|\, block_n \leftarrow data \;\; : |block_i| \in \{0,1\}^\ell$
For i = 1, . . . , n
  $input\_block \leftarrow chaining\_value \oplus block_i$
  $chaining\_value \leftarrow \mathsf{E}(K, input\_block)$
$output\_block \leftarrow chaining\_value$
Return $output\_block$

CTR-DRBG_df
Require: $input\_string$, $num\_bits \leq 512$
Ensure: req_bits
$L \leftarrow (\mathsf{len}(input\_string)/8)_{32}$; $N \leftarrow (num\_bits/8)_{32}$
$Z \leftarrow L \,\|\, N \,\|\, input\_string \,\|\, 0\text{x}80$
While $\mathsf{len}(Z) \bmod \ell \neq 0$
  $Z \leftarrow Z \,\|\, 0\text{x}00$
$temp \leftarrow \varepsilon$; $i \leftarrow 0$
$K \leftarrow \text{left}(0\text{x}000102...1D1E1F, \kappa)$
While $len(temp) < \kappa + \ell$
  $IV \leftarrow (i)_{32} \,\|\, 0^{\ell - 32}$
  $temp \leftarrow temp \,\|\, \mathsf{BCC}(K, (IV \,\|\, Z))$
  $i \leftarrow i + 1$
$K \leftarrow \text{left}(temp, \kappa)$
$X \leftarrow \text{select}(temp, \kappa + 1, \kappa + \ell)$
$temp \leftarrow \varepsilon$
While $len(temp) < num\_bits$
  $X \leftarrow \mathsf{E}(K, X)$
  $temp \leftarrow temp \,\|\, X$
$req\_bits \leftarrow \text{left}(temp, num\_bits)$
Return req_bits

Fig. 21: Algorithms setup, BCC and CTR-DRBG_df for CTR-DRBG.

G.2 Examples of Parameter Settings

	CTR-DRBG with df	CTR-DRBG w/out df	HMAC-DRBG	HASH-DRBG
Underlying Primitive	AES-128	AES-128	HMAC/SHA-256	SHA-256
Security strength	128	128	256	256
Output block len	128	128	256	256
Max no. of bits / request	2^{19}	2^{19}	2^{19}	2^{19}
Minimum len of *addin*	2^{32}	256	2^{32}	2^{32}
Max no. of requests between reseeds	2^{48}	2^{48}	2^{48}	2^{48}

Fig. 22: Table showing parameter settings for the NIST DRBGs described in Section 3. All quantities are given in bits.