An Analysis of NIST SP 800-90A

Joanne Woodage[1] and 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 NIST SP 800-90A [2]. The standard received considerable negative attention due to the controversy surrounding the now retracted DualEC-DRBG, which appeared 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 [13] 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 then conduct a more informal and practice-oriented exploration of flexibility in 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 these 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 [39, 32, 18, 6], 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.

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 [36]. This suspicion was confirmed by documents included in the Snowden leaks [29], leading to a revision of the document that removed the disgraced algorithm. The remaining DRBGs — which respectively use a hash function, HMAC, and a block cipher as their basic building blocks — are widely used. Indeed, any cryptographic software or hardware seeking FIPS certification *must* implement a PRNG from the standard [16, 38]. While aspects of these constructions have been analyzed [9, 20, 19, 35, 21, 33] and some implementation considerations discussed [5], these works make significant simplifying assumptions and / or treat certain algorithms rather than the constructions as a whole. To date, there has not 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 NIST SP 800-90A are 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 length (and sizable) outputs upon request, and support a variety of optional inputs and parameters — do not fit cleanly into the usual PRNG security models. With limited formal analysis 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 large parts of software relying on relatively unanalyzed algorithms.

Security claims. The standard claims that each of the NIST DRBGs is 'backtracking resistant' and 'prediction resistant'. The former property guarantees that in the event of a state compromise, prior output remains secure. The latter property ensures that if a compromised state is reseeded with sufficient entropy then security will be recovered. 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.

A number of factors may have contributed to this lack of analysis. It seems likely that the attention given to the Dual EC resulted in the other PRNGs in NIST SP 800-90A being comparatively overlooked. Secondly, our understanding of what a PRNG should achieve has developed since the NIST DRBGs were standardized in 2006. Indeed, the concept of robustness for PRNGs [13] was not formalized until 2013. Finally, the NIST DRBGs are based on fairly run-of-the mill concepts such as running a hash function in counter mode, and yet simultaneously display design quirks which significantly complicate analysis and defy attempts at a modular treatment. As such, perhaps they have escaped deeper analysis by not being 'interesting' enough to tackle for an attention-grabbing result and yet too tricky for a straightforward proof.

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 DRBGs in NIST SP 800-90A, 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 to be more constructive than modifying the constructions solely to derive a proof under weaker assumptions, and explain the rationale behind all such decisions.

Robustness proofs. Robustness, introduced by Dodis et al. [13], 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 if optional strings of additional input are not always included in next calls (see Section 3), then HMAC-DRBG is not forward secure. This contradicts the claimed backtracking resistance of HMAC-DRBG. This highlights the importance of formally proving security claims which at first sight 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 for 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 [17]. We must make various adaptations to accommodate the somewhat unorthodox interface of the NIST DRBGs, and specifying 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 with 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 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, each of the DRBGs 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 potentially recover strings of additional input which are fed to the DRBG and which may contain secrets. 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 and find that these potentially problematic implementation decisions are taken by implementors in the real world. We conclude with reflections and recommendations for the safe use of these DRBGs.

Related work. The PRNGs in NIST SP 800-90A have received little formal analysis to date; we provide an overview here. A handful of prior works have analyzed the NIST DRBGs as deterministic PRGs with an idealized initialization procedure. i.e., they prove the next algorithm 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 reality, state components must be derived from the entropy source using the setup algorithm. Such proofs are provided for CTR-DRBG by Campagna [9] and Shrimpton and Terashima [35], and for HMAC-DRBG by Hirose [19] and Ye et al. [21]. A formal verification of the mbedTLS implementation of HMAC-DRBG is also provided in [21]. None of these works model setup or reseeding; as far as we know, ours is the first analysis of these algorithms for HASH-DRBG and HMAC-DRBG. With the exception of [19], they do not model the use of additional input. Moreover, pseudorandomness is a weaker property than robustness and does not model state compromise. Kan [20] considers assumptions underlying the security claims of the DRBGs; however, the analysis is informal and contains inaccuracies. To our knowledge, this is the only previous work to consider HASH-DRBG. Ruhault claims a potential attack against the robustness of CTR-DRBG [33]. However, the specification of CTR-DRBG's BCC function in [33] is different to that provided by the standard. In [33], 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 $[B_1, \ldots, B_n]$. The attack from [33] does not work when the correct BCC function is used, and it does not seem possible to recover the attack.

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 write $\{0,1\}^{n \leq i \leq \overline{n}}$ to denote the set of binary strings of length between n and $\overline{n}$-bits inclusive. 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 their concatenation. 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 $\mathbb{N} = \{1, 2, \dots\}$ denote the set of natural numbers, and let $\mathbb{N}^{\leq n} = \{1, 2, \dots, n\}$.

Entropy and Cryptographic Components. In the full version, 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.

PRNGs with input. A *pseudorandom number generator with input* (PRNG) [13] produces pseudorandom bits and offers strong security guarantees (see Section 4) when given continual access to an imperfect source of randomness. We define PRNGs, and discuss our choice of syntax, below.

Definition 1. *A PRNG with input is a tuple of algorithms* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *with associated parameters* $(p, \overline{p}, \alpha, \beta_{max})$*, defined:*

- setup : $\mathrm{Seed} \times \{0,1\}^{p \leq i \leq \overline{p}} \times \mathcal{N} \rightarrow \mathcal{S}$ *takes as input a seed* $X \in \mathrm{Seed}$*, an entropy sample* $I \in \{0,1\}^{p \leq i \leq \overline{p}}$*, and a nonce* $N \in \mathcal{N}$ *(where* Seed*,* $\mathcal{N}$*, and* $\mathcal{S}$ *denote the seed space, nonce space, and state space of the PRNG respectively), and returns an initial state* $S_0 \in \mathcal{S}$*.*
- $\mathsf{refresh}$: $\mathrm{Seed} \times \mathcal{S} \times \{0,1\}^{p \leq i \leq \overline{p}} \rightarrow \mathcal{S}$ *takes as input a seed* $X \in \mathrm{Seed}$*, a state* $S \in \mathcal{S}$*, and an entropy sample* $I \in \{0,1\}^{p \leq i \leq \overline{p}}$*, and returns a state* $S' \in \mathcal{S}$*.*
- next : $\mathrm{Seed} \times \mathcal{S} \times \mathbb{N}^{\leq \beta_{max}} \times \{0,1\}^{\leq \alpha} \rightarrow \{0,1\}^{\leq \beta_{max}} \times \mathcal{S}$ *takes as input a seed* $X \in \mathrm{Seed}$*, a state* $S \in \mathcal{S}$*, a parameter* $\beta \in \mathbb{N}^{\leq \beta_{max}}$*, and a string of additional input* $addin \in \{0,1\}^{\leq \alpha}$*, 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 seed or additional input is never used), then we omit these parameters.*

Discussion. Our definition follows that of Dodis et al. [13], 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. [34], 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 a random seed 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 the full version. NIST SP 800-90A 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

NIST SP 800-90A defines three DRBG mechanisms, HASH-DRBG, HMAC-DRBG, and CTR-DRBG. The former two are based on an approved hash function (e.g., SHA-256), and the latter on an approved block cipher (e.g., AES-128); we include parameters for these variants in the full version.

Algorithms. The standard specifies (Instantiate, Reseed, Generate) algorithms for each of the DRBGs. These map directly into the (setup, refresh, next) algorithms in the PRNG model of Definition 1. Since the NIST DRBGs are not specified to take a seed (see Section 4), we take $X = \varepsilon$ and Seed $= \emptyset$ in this mapping, and omit these parameters from subsequent definitions. For consistency, we refer to the NIST DRBG algorithms as (setup, refresh, next) throughout. These (setup, refresh, next) algorithms underly (respectively) the (`Instantiate`, `Reseed`, `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 β_{max}), and return an error if these checks fail. If not, the function fetches the internal state of the DRBG, plus any other required inputs (e.g., entropy samples, a nonce, etc.), and the underlying algorithm is applied to these inputs. The resulting outputs are returned and / or used to update the internal state, and the successful status of the call is indicated to the caller. To avoid cluttering our exposition we abstract away this process, assuming that required inputs are provided to algorithms (without modeling how these are fetched), and that all inputs and requests are valid, omitting success / error notifications.

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 e.g., the security strength of the instantiation. We typically omit administrative information as this shall be clear from the context.

Entropy sources. A DRBG must have access to an *approved entropy source*[3] during initial state generation via setup, after which the DRBG is said to be *instantiated*. The function `Get_entropy_input`() is used to request an entropy sample I of length within range $[p, \overline{p}]$ containing a given amount of entropy (discussed further below). Nonces used by setup 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.

Reseeding. Entropy samples drawn from the source are periodically incorporated into the DRBG state via refresh. A parameter *reseed_interval* indicates the maximum number of output generation requests allowed by an implementation before a reseed is forced; this is tracked by a state component called a reseed counter (*cnt*). The reseed counter is not a security critical state variable, and we assume that it is publicly known. For all approved DRBG variants (except CTR-DRBG based on 3-KeyTDEA) *reseed_interval* may be as large as 2^{48}. We assume here that reseeds are always explicitly requested by the caller; this is without loss of generality. A DRBG instantiation is parameterized by a *security*

[3] Either a live entropy source as approved in NIST SP 800-90B [37] or a (truly) random bit generator as per NIST SP 800-90C [3].

strength $\gamma^* \in \{112, 128, 192, 256\}$, where the highest supported security strength depends on the underlying primitive. Each entropy sample used by setup and refresh must contain at least γ^*-bits of entropy[4].

Output generation. The caller may request outputs of length up to β_{max} bits via the input β to the next function. For all approved DRBG variants (except CTR-DRBG based on 3-KeyTDEA), β_{max} may be as large as 2^{19}.

Additional input. Optional strings of additional input (denoted *addin*) may 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. Optional additional input may also be provided in refresh calls and during setup; for brevity, we do not model this here and omit these inputs from the presentation of the algorithms.

3.2 HASH-DRBG.

HASH-DRBG is built from an approved 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 the full version.

Algorithms. The component algorithms for HASH-DRBG are shown in Figure 2. Both setup and refresh 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 first incorporates any additional input into the counter V (lines 3 - 5). Output blocks are then produced by hashing V in CTR-mode (lines 7 - 10). At the conclusion of the call, 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 (lines 12 - 13).

3.3 HMAC-DRBG

HMAC-DRBG is based on $\mathsf{HMAC} : \{0,1\}^{\ell} \times \{0,1\}^{\leq \omega} \to \{0,1\}^{\ell}$, built from an approved hash function. 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 security critical.

Algorithms. The component algorithms of HMAC-DRBG are shown in Figure 2. Algorithms setup and refresh both use the update subroutine to incorporate an entropy sample I into K and V. For setup, these variables are initialized to $K = \mathtt{0x00} \ldots 00$ and $V = \mathtt{0x01} \ldots 01$ prior to this process. Output production via next first incorporates any additional input into K and V via the update

[4] In contrast, robustness [13] requires that a PRNG is secure when reseeded with a set of entropy samples which *collectively* has γ^*-bits of entropy. Looking ahead to Section 5, we analyze HASH-DRBG with respect to this stronger notion.

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

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

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

```
HASH-DRBG next
Require: $S = (V, C, cnt), \beta, addin$
Ensure: $R, S' = (V', C', cnt')$
If $cnt > reseed\_interval$
    Return reseed_required
If $addin \neq \varepsilon$
    $w \leftarrow \mathsf{H}(\mathtt{0x02} \,\|\, V \,\|\, addin)$
    $V \leftarrow (V + w) \bmod 2^L$
$data \leftarrow V$ ; $temp_R \leftarrow \varepsilon$ ; $n \leftarrow \lceil \beta/\ell \rceil$
For $j = 1, \ldots, n$
    $r \leftarrow \mathsf{H}(data)$
    $data \leftarrow (data + 1) \bmod 2^L$
    $temp_R \leftarrow temp_R \,\|\, r$
$R \leftarrow \mathrm{left}(temp_R, \beta)$
$H \leftarrow \mathsf{H}(\mathtt{0x03} \,\|\, V)$
$V' \leftarrow (V + H + C + cnt) \bmod 2^L$
$C' \leftarrow C$ ; $cnt' \leftarrow cnt + 1$
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 $j = 1, \ldots, m$
    $V \leftarrow (V + 1) \bmod 2^{\ell}$ ; $Z \leftarrow \mathsf{E}(K, V)$
    $temp \leftarrow temp \,\|\, Z$
$temp \leftarrow \mathrm{left}(temp, (\kappa + \ell))$
$temp \leftarrow temp \oplus provided\_data$
$K \leftarrow \mathrm{left}(temp, \kappa)$
$V \leftarrow \mathrm{right}(temp, \ell)$
Return $K, V$
```

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

```
HMAC-DRBG setup
Require: $I,\ N$
Ensure: $S_0 = (K_0, V_0, cnt_0)$
$seed\_material \leftarrow I \,\|\, N$
$K \leftarrow \mathtt{0x00} \ldots 00$
$V \leftarrow \mathtt{0x01} \ldots 01$
$(K_0, V_0) \leftarrow \mathsf{update}(seed\_material, K, V)$
$cnt_0 \leftarrow 1$
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: $R, S' = (K', V', cnt')$
If $cnt > reseed\_interval$
    Return reseed_required
If $addin \neq \varepsilon$
    $(K, V) \leftarrow \mathsf{update}(addin, K, V)$
$temp \leftarrow \varepsilon$ ; $n \leftarrow \lceil \beta/\ell \rceil$
For $j = 1, \ldots, n$
    $V \leftarrow \mathsf{HMAC}(K, V)$
    $temp \leftarrow temp \,\|\, V$
$R \leftarrow \mathrm{left}(temp, \beta)$
$(K', V') \leftarrow \mathsf{update}(addin, K, V)$
$cnt' \leftarrow cnt + 1$
Return $R, (K', V', cnt')$
```

```
CTR-DRBG next
Require: $S = (K, V, cnt), \beta, addin$
Ensure: $R, S' = (K', V', cnt')$
If $cnt > reseed\_interval$
    Return reseed_required
If $addin \neq \varepsilon$
    If derivation function used then
        $addin \leftarrow \mathsf{CTR\text{-}DRBG\_df}(addin, (\kappa + \ell))$
    Else if $len(addin) < (\kappa + \ell)$ then
        $addin \leftarrow addin \,\|\, 0^{(\kappa + \ell - len(addin))}$
    $(K, V) \leftarrow \mathsf{update}(addin, K, V)$
Else $addin \leftarrow 0^{\kappa + \ell}$
$temp \leftarrow \varepsilon$ ; $n \leftarrow \lceil \beta/\ell \rceil$
For $j = 1, \ldots, n$
    $V \leftarrow (V + 1) \bmod 2^{\ell}$ ; $r \leftarrow \mathsf{E}(K, V)$
    $temp \leftarrow temp \,\|\, r$
$R \leftarrow \mathrm{left}(temp, \beta)$
$(K', V') \leftarrow \mathsf{update}(addin, K, V)$
$cnt' \leftarrow cnt + 1$
Return $R, (K', V', cnt')$
```

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

function (lines 3 - 4). Output blocks r are generated by iteratively computing $V \leftarrow \mathsf{HMAC}(K, V)$ and setting $r = V$ (lines 6 - 8). At the conclusion of the call, both key and counter are updated via the update function (line 10).

3.4 CTR-DRBG

CTR-DRBG is built from an approved block cipher $\mathsf{E} : \{0,1\}^\kappa \times \{0,1\}^\ell \rightarrow \{0,1\}^\ell$. The working state is defined $S = (K, V, cnt)$, where key $K \in \{0,1\}^\kappa$ and counter $V \in \{0,1\}^\ell$ are the security critical state variables.

Algorithms. Component algorithms for CTR-DRBG are shown in Figure 2[5]. Use of the derivation function CTR-DRBG_df is optional only if the implementation has access to a 'full entropy source' which returns statistically close to uniform strings. Output generation via next first incorporates any additional input *addin* into the state via the update function (line 8). If a derivation function is used, additional input *addin* is conditioned into a $(\kappa + \ell)$-bit string with CTR-DRBG_df prior to this process (line 5); otherwise *addin* 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).

4 Robustness in the Random Oracle Model

The security properties of backtracking and prediction resistance claimed in the standard have never been formally investigated. We address this be analyzing the robustness [13] of HASH-DRBG and HMAC-DRBG. This models a powerful attacker who is able to compromise the state and influence the entropy source of the PRNG, and is easily verifed to imply both backtracking and prediction resistance. In this section, we adapt the robustness model of [13] to accommodate the NIST DRBGs, and 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* [13]. Formally, a distribution sampler $\mathcal{D} : \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^* \times \mathbb{R}^{\geq 0} \times \{0,1\}^*$ 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 $\gamma_1 \geq \gamma^*$. Here, condition **(2)** extends the definition of [13] to model the sample (which recall must contain γ^* bits of entropy) with which the DRBG is initially seeded during

[5] We do not directly analyze setup, refresh and CTR-DRBG_df in this work, and so defer their presentation to the full version.

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[6].

4.1 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. [13] to the ROM.

Robustness. Consider game Rob shown in Figure 3. The game is parameterized by an entropy threshold γ^*. We expect security when the entropy in the system exceeds this value. When analyzing the NIST DRBGs, we take γ^* to be the security strength of the instantiation. 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 (indicated in superscript e.g., $\mathsf{setup}^{\mathsf{H}}$). For reasons discussed below, we do *not* give the sampler $\mathcal{D}$ access to H. To the best of our knowledge, this is the first treatment of robustness in the ROM, and our model may be useful beyond analyzing HASH-DRBG and HMAC-DRBG. We additionally modify game Rob from [13] to: **(1)** accommodate our PRNG syntax (including the use of additional input, discussed below); **(2)** remove the Next oracle, which was shown in [11] to be without loss of generality; and **(3)** generate the initial state with the deterministic $\mathsf{setup}^{\mathsf{H}}$ algorithm using the first entropy sample output by the sampler, similarly to [34].

The game is implicitly parameterized by a nonce distribution $\mathcal{N}$, 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 that $\mathcal{N}$ is public and the nonce used during initalization is provided to $\mathcal{A}$. Similarly, we assume the attacker can choose the strings of additional input which may be included in next calls. These are conservative assumptions, since any entropy in these values can only make the attacker's job harder. With this in place, the Rob advantage of an adversary $\mathcal{A}$, and a $(q_{\mathcal{D}}^{+}, \gamma^*)$-legitimate sampler $\mathcal{D}$, is defined

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

We say that $\mathcal{A}$ is a $(q_{\mathsf{H}}, q_R, q_{\mathcal{D}}, q_C, q_S)$-adversary if it makes q_{H} queries to the random oracle H, q_R queries to its RoR oracle, a total of q_S queries to its Get and Set oracles, and $q_{\mathcal{D}}$ queries to its Ref oracle of which at most q_C are consecutive.

Fixed length variant. While we define the general game Rob here, our robustness proofs will be in a slightly restricted variant Rob_{β}, in which the attacker may only request outputs of fixed length $\beta \leq \beta_{max}$ in RoR queries. This simplifying assumption is to avoid further complicating bounds with parameters

[6] NIST SP 800-90B [37] defines the entropy estimate of sample I as $\mathrm{H}_{\infty}(I)$, rather than this conditioned on other samples and associated data. However, since the tests in NIST SP 800-90B estimate entropy using multiple samples drawn from the source, it seems reasonable to assume the conditional entropy condition is satisfied also.

indicating the length of each RoR query. Results for the general game Rob can be recovered as a straightforward extension of our proofs.

Standard model forward security. Our negative result about the forward security of HMAC-DRBG shall be in the standard model. We define game Fwd to be a restricted variant of Rob in which: **(1)** we remove oracle access to H from all algorithms; and **(2)** the attacker $\mathcal{A}$ is allowed no Set queries, and makes a single Get query after which they may make no further queries. The forward security advantage of $(\mathcal{A}, \mathcal{D})$ is defined

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

The problem of seeding. Since deterministic extraction from imperfect sources is impossible in general, the PRNG in game Rob is initialized with a random public seed X which crucially is independent of the entropy source. Unfortunately (for our analysis), none of the NIST DRBGs are specified to take a seed, (i.e., $X = \varepsilon$ in our modeling). Moreover, all state components and inputs to HASH-DRBG and HMAC-DRBG may depend on the entropy source, and so cannot be reframed as a seed without adding substantial assumptions; we discuss this further in the full version. At this point we are faced with two choices. We either: **(1)** give sampler $\mathcal{D}$ access to H (as in the robustness in the IPM model of [17]), and either modify the NIST DRBGs to accommodate a random seed or restrict our analysis to implementations for which additional input is sufficiently independent of the source to suffice as a seed. Or: **(2)** do not give $\mathcal{D}$ access to H. In this case, the oracle H with respect to which security analysis is carried out is chosen randomly and independently of the entropy source, and so serves the same purpose as a seed. We take the latter approach for a number of reasons. Firstly, we wish to analyze the NIST DRBGs as they are specified and used, and so modifying the construction or restricting the implementations we can reason about (as per **(1)**) solely to facilitate the analysis seems counterproductive. Secondly, as pointed out in [35], generating a seed is challenging in practice due to the necessary independence from the entropy source. Moreover, given the litany of tests which approved entropy sources in NIST SP 800-90B are subjected to, it seems reasonable to assume that the pathological sources used to illustrate e.g., deterministic extraction impossibility results, are unlikely to pass these tests.

```
$\mathrm{Rob}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\gamma^*}$
$\mathsf{H} \leftarrow\!\!\$\ \mathcal{H}$ ; $b \leftarrow\!\!\$\ \{0,1\}$ ; $N \leftarrow \mathcal{N}$
$\sigma \leftarrow \varepsilon$ ; $X \leftarrow\!\!\$\ \mathrm{Seed}$
$(\sigma, I, \gamma, z) \leftarrow\!\!\$\ \mathcal{D}(\sigma)$
$S \leftarrow \mathsf{setup}^{\mathsf{H}}(X, I, N)$
$c \leftarrow \gamma^*$ ; $\overline{\gamma} \leftarrow (\gamma, z, N)$
$b^* \leftarrow\!\!\$\ \mathcal{A}^{\mathrm{Ref,RoR,Get,Set,}\mathsf{H}}(X, \overline{\gamma})$
Return $(b = b^*)$
```

```
Ref
$(\sigma, I, \gamma, z) \leftarrow\!\!\$\ \mathcal{D}(\sigma)$
$S \leftarrow \mathsf{refresh}^{\mathsf{H}}(X, S, I)$
$c \leftarrow c + \gamma$
Return $(\gamma, z)$

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

```
RoR($\beta$, $addin$)
$(R_0, S) \leftarrow \mathsf{next}^{\mathsf{H}}(X, S, \beta, addin)$
If $c < \gamma^*$
    Return $R_0$
    $c \leftarrow 0$
Else $R_1 \leftarrow\!\!\$\ \{0,1\}^{\beta}$
Return $R_b$
```

```
Get
Return $S$
$c \leftarrow 0$

Set($S^*$)
$S \leftarrow S^*$
$c \leftarrow 0$
```

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

Security games. A key insight of [13] is that the complex notion of robustness can be decomposed into two simpler notions called preserving and recovering security. The former models the PRNG's ability to maintain security if the state is secret but the attacker is able to influence the entropy source. The latter models the PRNG's ability to recover from state compromise after sufficient (honestly generated) entropy has entered the system. Here we will utilize the variants of these from [34], which extended the original definitions and added a new game Init modelling initial state generation.

Consider games Pres, Rec, and Init shown in Figure 4, given here for Rob_{β}. Here we have adapted the notions of [34] in the natural way to accommodate: **(1)** a random oracle; and **(2)** our PRNG syntax. It is straightforward to extend our analysis to accommodate variable length outputs. All games are defined with respect to a *masking function*, which is a randomized function $\mathsf{M} : \mathcal{S} \cup \{\varepsilon\} \rightarrow \mathcal{S}$ where $\mathcal{S}$ denotes the state space of the PRNG. Here, we extend the definition of [34] to include ε in the domain (implicitly assuming 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). We discuss the reasons for this adaptation in Section 5. We give the masking function access to the random oracle, indicated by M^{H}. We make a number of 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 [34]. 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^* which is *output* by $\mathsf{next}^{\mathsf{H}}$. Finally, in Pres, we allow $\mathcal{A}$ to output $S \in \mathcal{S} \cup \{\varepsilon\}$ at the start of his challenge rather than $S \in \mathcal{S}$. In all cases, this is to accommodate the somewhat complicated state distribution of $\mathsf{HASH\text{-}DRBG}$ (see Section 5). For all $\mathrm{Gm}_y^x \in \{\mathrm{Init}_{\mathcal{G},\mathsf{M},q_{\mathcal{D}},\gamma^*}^{\mathcal{A},\mathcal{D}}, \mathrm{Pres}_{\mathcal{G},\mathsf{M},\beta}^{\mathcal{A},}, \mathrm{Rec}_{\mathcal{G},\mathsf{M},\gamma^*,q_{\mathcal{D}},\beta}^{\mathcal{A},\mathcal{D}}\}$ we define

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

An adversary in Init is said to be a q_{H}-adversary if it makes q_{H} queries to its H oracle. An adversary in game Pres or Rec is said to be a (q_{H}, q_C)-adversary if it makes q_{H} queries to its H oracle and always outputs $d \leq q_C$.

With this in place, the following theorem — which says that Init, Pres and Rec security collectively imply Rob security — is an adaptation of the analogous results from [34], [17]. 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 [34] that masking functions possess a property called idempotence. The proof is given in the full version. We note that the original result [13] omits a factor of two from the right-hand side of the equation which we recover here.

Theorem 1. *Let* $\mathcal{G} = (\mathsf{setup}^{\mathsf{H}}, \mathsf{refresh}^{\mathsf{H}}, \mathsf{next}^{\mathsf{H}})$ *be a PRNG with input with associated parameter set* $(p, \overline{p}, \alpha, \beta_{max})$*, 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* $\mathsf{M}^{\mathsf{H}} : \mathcal{S} \cup \{\varepsilon\} \rightarrow \mathcal{S}$ *be a* *masking function for which each invocation of* M^{H} *makes at most* q_{M} H *queries. Then for any* $(q_{\mathsf{H}}, q_R, q_{\mathcal{D}}, q_C, q_S)$*-adversary* $\mathcal{A}$ *and* $(q_{\mathcal{D}}^{+}, \gamma^*)$*-legitimate sampler* $\mathcal{D}$ *in game* Rob_β *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* $(q_{\mathsf{H}} + q_{\mathcal{D}} \cdot q_{ref} + q_R \cdot (q_{nxt} + q_{\mathsf{M}}), q_C)$*-adversaries* $\mathcal{A}_2, \mathcal{A}_3$*, such that*

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

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 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 [13, 34, 17]. Developing a technique to obtain tighter bounds is an important open question.

```
$\mathrm{Init}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*,q_{\mathcal{D}}}$
$\mathsf{H} \leftarrow\$ \mathcal{H}$ ; $b \leftarrow\$ \{0,1\}$ ; $N \leftarrow \mathcal{N}$
$\sigma_0 \leftarrow \varepsilon$ ; $X \leftarrow\$$ Seed
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 $(b = b^*)$
```

```
$\mathrm{Pres}^{\mathcal{A}}_{\mathcal{G},\mathsf{M},\beta}$
$\mathsf{H} \leftarrow\$ \mathcal{H}$ ; $b \leftarrow\$ \{0,1\}$
$X \leftarrow\$$ Seed
$(S_0', I_1, \ldots, I_d, addin) \leftarrow\$ \mathcal{A}^{\mathsf{H}}(X)$
If $S_0' \notin \mathcal{S} \cup \{\varepsilon\}$ return $\perp$
$S_0 \leftarrow\$ \mathsf{M}^{\mathsf{H}}(S_0')$
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 $(b = b^*)$
```

```
$\mathrm{Rec}^{\mathcal{A},\mathcal{D}}_{\mathcal{G},\mathsf{M},\gamma^*,q_{\mathcal{D}},\beta}$
$\mathsf{H} \leftarrow\$ \mathcal{H}$ ; $b \leftarrow\$ \{0,1\}$
$\sigma \leftarrow \varepsilon$ ; $X \leftarrow\$$ Seed ; $\mu \leftarrow 1$
For $k = 1, \ldots, q_{\mathcal{D}} + 1$
   $(\sigma_k, I_k, \gamma_k, z_k) \leftarrow\$ \mathcal{D}(\sigma_{k-1})$
$(S_0, d, addin) \leftarrow\$ \mathcal{A}^{\mathsf{H},\mathrm{Sam}}(X, (\gamma_k, z_k))_{i=1}^{q_{\mathcal{D}}+1})$
If $S_0 \notin \mathcal{S}$ return $\perp$
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 $(R^*, S^*) \leftarrow\$ \mathsf{next}^{\mathsf{H}}(X, S_d, \beta, addin)$
Else $R^* \leftarrow \{0,1\}^\beta$ ; $S^* \leftarrow\$ \mathsf{M}^{\mathsf{H}}(S_d)$
$b^* \leftarrow\$ \mathcal{A}(X, R^*, S^*, (I_k)_{k>\mu+d})$
Return $(b = b^*)$
```

```
Sam()
$\mu = \mu + 1$
Return $I_\mu$
```

```
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 $\mathsf{M} \ : \ \mathcal{S} \cup \{\varepsilon\} \to \mathcal{S}$.

5 Analysis of HASH-DRBG

We now present our analysis of the robustness of HASH-DRBG, in which the underlying hash function $\mathsf{H} \ : \ \{0,1\}^{\leq \omega} \to \{0,1\}^\ell$ is modelled as a random oracle. Our proof is with respect to the masking function M^{H} shown in Figure 5.

To avoid further complicating security bounds we assume that HASH-DRBG is never called with additional input; we expect extending the proof to include additional input to be straightforward.

Challenges. Certain features of HASH-DRBG significantly complicate the proof, and necessitated adaptations in our security modeling (Section 4). 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 model this in games Init, Pres and Rec, we extended the domain of M to include the empty string ε to indicate that an idealized state of the first form should be returned (for example, when modeling initial state generation in Init), and extended Pres to allow $\mathcal{A}$ to output ε at the start of the challenge (which is required for the proof of Theorem 1, see the full version). Juggling these different state distributions complicates analysis, and introduces multiple cases into the proof of Pres security. Care is also required when analyzing the distribution of $S' = \mathsf{M}^{\mathsf{H}}(S)$ for $S \in \mathcal{S}$, which idealizes the distribution of the state $S' = (V', C', cnt')$ as updated following an output generation request. It is straightforward to verify that V' is distributed uniformly over the range $[V + C + cnt, V + C + cnt + (2^{\ell} - 1)]$ where $S = (V, C, cnt)$ and $L > \ell + 1$. To accommodate this dependency between the updated state S' and the previous state S, we have modifed games Pres and Rec so that it is S which is masked instead of S'. 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. We discuss these issues further in the full version.

```
$\mathsf{M}^{\mathsf{H}}(S)$
If $S = \varepsilon$
    $V' \leftarrow\!\!\$\ \{0,1\}^L$
    $C' \leftarrow \mathsf{HASH\text{-}DRBG\_df}^{\mathsf{H}}(0\mathrm{x}00 || V, L)$
    $cnt' \leftarrow 1$
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 $S'$
```

Fig. 4: Masking function for proof of Theorem 2

Parameter settings. We provide a general treatment into which any parameter setting may be slotted, subject to two restrictions which are utilized in the proof. Namely, we assume that $L > \ell + 1$ and $n < 2^L$, where $n = \lceil \beta/\ell \rceil$ is the number of output blocks produced by $\mathsf{next}^{\mathsf{H}}$ to satisfy a request for β-bits (without this latter restriction HASH-DRBG is trivially insecure, as the same counter would be hashed twice during output production). 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 an 8-bit string, respectively, by HASH-DRBG_df. All approved hash functions fall well within these parameters. (Indeed, for all of these, $L > 2\ell$, $n < 3277 \ll 2^L$, and $m \leq 3$.)

Robustness. With this in place, we present the following theorem bounding the Rob security of HASH-DRBG. The proof follows from a number of lemmas presented below, combined with Theorem 1. (When applying Theorem 1, it is readily verified that for $\mathsf{HASH\text{-}DRBG}$ $q_{nxt} = n+1$, $q_{ref} = 2m$, and $q_{\mathsf{M}} = m$.) The proofs of all lemmas are given in the full version.

Theorem 2. *Let $\mathcal{G}$ be* $\mathsf{HASH\text{-}DRBG}$ *with parameters $(p, \overline{p}, \alpha, \beta_{max})$, built from a hash function $\mathsf{H} : \{0,1\}^{\leq \omega} \rightarrow \{0,1\}^{\ell}$ which we model as a random oracle. Let L denote the state length of* $\mathsf{HASH\text{-}DRBG}$ *where $L > \ell + 1$, $n = \lceil \beta/\ell \rceil$, and $m = \lceil L/\ell \rceil$. Let M^{H} denote the masking function shown in Figure 5, and suppose that* $\mathsf{HASH\text{-}DRBG}$ *is never called with additional input. Then for any $(q_{\mathsf{H}}, q_R, q_{\mathcal{D}}, q_C, q_S)$-attacker $\mathcal{A}$ in game* Rob_β *against $\mathcal{G}$, and any $(q_{\mathcal{D}}^{+}, \gamma^*)$-legitimate sampler $\mathcal{D}$, it holds that*

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

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 first argue that the states returned by $\mathsf{setup}^{\mathsf{H}}$ are indistinguishable from $\mathsf{M}^{\mathsf{H}}(\varepsilon)$. The $q_{\mathsf{H}} \cdot 2^{-\gamma^*}$ term follows since the initial state variable V_0 will be indistinguishable from random unless the attacker queries H on one of the points which was hashed to produce it. This in turn requires $\mathcal{A}$ to guess the value of the entropy sample I_1, which contains γ^*-bits of entropy. 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 $I_1 || N = 0\text{x}00 || V_0$ where I_1 and N denote the entropy input and nonce (an event which — while very unlikely — is not precluded by the parameter constraints in the standard), then the derived values of V_0 and C_0 will be equal, allowing the attacker to distinguish with high probability. A small tweak to the design of setup (e.g., prepending 0x01 to $I || N$ before hashing) would have avoided this. For implementations of $\mathsf{HASH\text{-}DRBG}$ for which such a collision is impossible (e.g., due to length restrictions on the input I) this additional term can be removed.

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

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

Pres security. At the start of game Pres, the (q_{H}, q_C)-attacker $\mathcal{A}$ outputs $(S'_0, I_1, \ldots, I_d)$ where $d \leq q_C$. 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 first argues that unless $\mathcal{A}$ queries H on the counter V_0, or any of the counters $V_1, \ldots, V_{d-1}$ passed through during reseeding, then (barring certain accidental collisions) $S_d = (V_d, C_d, cnt_d)$ is indistinguishable from a masked state. The proof then shows that, unless the attacker can guess V_d, the resulting output / state pair is indistinguishable from its idealized counterpart. 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* (q_{H}, q_C)*-adversary* $\mathcal{A}$ *in game* Pres *against* $\mathcal{G}$*, it holds that*

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

Rec security. The first stage 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 (see full version). Our proof is closely based on the analogous result for sponge-based PRNGs in the ideal permutation model (IPM) of Gazi and Tessaro [17], 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 handle the state component C and the more involved reseeding process, which concatenates the responses to multiple H queries to derive updated counters. With this in place, an analogous argument to that made for Pres security implies that an output / state pair produced by applying $\mathsf{next}^{\mathsf{H}}$ to this masked state is indistinguishable from its idealized counterpart.

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

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

Using Theorem 1 to combine Lemmas 1, 2 and 3 — which bound the Init, Pres and Rec security of HASH-DRBG respectively — proves Theorem 2, and completes our analysis of the robustness of HASH-DRBG in the ROM.

6 Analysis of HMAC-DRBG

In this section, we 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 additional input is provided in next calls.

6.1 Negative Result: HMAC-DRBG Called Without Additional Input is Not Forward Secure

We present an attack which breaks the forward security of HMAC-DRBG if called without additional input. This contradicts the claim in the standard that

HMAC-DRBG is backtracking resistant. Since Rob security implies Fwd security, this rules out a proof of robustness in this case also.

The attack. Consider the application of update at the conclusion of a next call for HMAC-DRBG (Figure 2). Notice that if $addin = \varepsilon$, then the final two lines of update are not executed. In this case, 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. An attacker $\mathcal{A}$ in game Fwd who makes a RoR query with $addin = \varepsilon$ 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. We note that the observation that V^* depends on r^* is also implicit in the proof of pseudorandomness by Hirose [19]; however, the connection to forward security is not made in that work. To concretely bound $\mathcal{A}$'s advantage we define game $\text{Fwd}^{\$}$, which is identical to game Fwd against HMAC-DRBG except the PRNG is initialized with an 'ideally distributed' state $S_0 = (K_0, V_0, cnt_0)$ where $K_0, V_0 \leftarrow\!\!{\scriptstyle\$}\ \{0,1\}^\ell$ and $cnt_0 \leftarrow 1$. The attacker's job can only be harder in $\text{Fwd}^{\$}$, since they cannot exploit any flaws in the setup procedure.

Theorem 3. *Let $\mathcal{G}$ be* HMAC-DRBG *built from the function* $\mathsf{HMAC} : \{0,1\}^\ell \times \{0,1\}^{\leq \omega} \to \{0,1\}^\ell$*, with parameters $(p, \overline{p}, \alpha, \beta_{max})$ such that $\beta_{max} \geq \ell$. Then there exist efficient adversaries $\mathcal{A}, \mathcal{B}$, such that for any sampler $\mathcal{D}$, it holds that*

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

$\mathcal{A}$ makes one RoR query in which additional input is not included, 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.

Discussion. The first negative term on the right-hand side of the above equation is the advantage of an attacker $\mathcal{B}$ who tries to break the PRF-security of HMAC given two queries to its real-or-random function oracle; since HMAC is widely understood to be a secure PRF, we expect this term to be small. Similarly, ℓ denotes the output length of HMAC and so the second negative term will be small for all commonly used hash functions. This implies that $\mathcal{A}$ succeeds with probability close to one, making this an effective attack. For simplicity, the theorem assumes that outputs of length ℓ bits may be requested (i.e., $\beta_{max} \geq \ell$); in the full version, we discuss how to relax this restriction.

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

In this section, we prove that HMAC-DRBG is robust in the ROM when additional input is always used, with respect to a restricted (but realistic) class of samplers. We model the function $\mathsf{HMAC} : \{0,1\}^\ell \times \{0,1\}^{\leq \omega} \to \{0,1\}^\ell$ 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.

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

Fig. 5: Masking function for proof of Theorem 4.

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 unclear. These results require showing that HMAC is a good randomness extractor. In games Init and Rec, the HMAC key is chosen by or known to the attacker, and so we cannot appeal to PRF-security. 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 to be inherently required. The extraction properties of HMAC (under various idealized assumptions) were studied in [12]. However, these consider a single-use version of extraction which is weaker than what is required here, and typically require inputs containing much more entropy than is required by the standard, and so are not generally applicable to real-world implementations of HMAC-DRBG.

By modeling HMAC as a keyed RO, we can analyze HMAC-DRBG 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 other works in which HMAC is used with a known key or to extract from lower entropy sources e.g., [32, 23, 24]. In [14], 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 [31]).

Discussion. A standard model proof of Rec security for HMAC-DRBG would be a stronger and more satisfying result. While 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 [19, 21] omit reseeding and initialization, assuming HMAC-DRBG is initialized with a state for which $K, V \leftarrow\$ \{0,1\}^{\ell}$, which is far removed from HMAC-DRBG in a real system. Our work is also the first to consider security properties stronger than pseudorandomness. We hope our result is a valuable first step to progress the understanding of this widely deployed (yet little analyzed) PRNG, and a useful starting point for further work to extend.

Sampler. Our proof is 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 $i \in [1, q_{\mathcal{D}} + 1]$ (i.e., each sample I contains γ^*-bits of entropy). This is a simplifying assumption, making the proof of Rec security less complex. However, we stress that this is the entropy level per sample required by the standard, and so this is precisely the restriction imposed on allowed entropy sources. An H-coefficient analysis, as in the proof of Lemma 3, seems likely to yield a fully general result.

Proof of robustness. We now present the following theorem bounding the robustness of HMAC-DRBG. The proof follows from the lemmas presented below, combined with Theorem 1. (It is straightforward to verify that $q_{ref} = 4$, $q_{nxt} = n + 8$, and $q_{\mathsf{M}} = 0$). The proofs of all lemmas are given in the full version.

Theorem 4. *Let $\mathcal{G}$ be* HMAC-DRBG *with parameters $(p, \overline{p}, \alpha, \beta_{max})$, built from* HMAC $: \{0,1\}^{\ell} \times \{0,1\}^{\leq \omega} \rightarrow \{0,1\}^{\ell}$ *which we model as a keyed random oracle. Let $\mathsf{M}^{\mathsf{HMAC}}$ be the masking function shown in Figure 6 and $n = \lceil \beta/\ell \rceil$. Then for any $(q_{\mathsf{H}}, q_R, q_{\mathcal{D}}, q_C, q_S)$-attacker $\mathcal{A}$ in game* Rob$_\beta$ *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*

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

Here $\epsilon_1 = 12q_C + 10 + (4q_C - 2) \cdot 2^{-\gamma^*}$, $\epsilon_2 = (q_C \cdot (10q_C + 4n + 18 + (q_C - 1) \cdot 2^{-(\gamma^*-1)}) + 6n + 16)$, $\epsilon_3 = n(n+1)$, and $\bar{q}_{\mathsf{H}} = (q_{\mathsf{H}} + 4 \cdot q_{\mathcal{D}} + (n+8) \cdot q_R)$.

Concrete example. For HMAC-SHA-512, $\ell = 512$ and the bound is dominated by the $\mathcal{O}(\bar{q}_{\mathsf{H}} \cdot q_R) \cdot 2^{-(\gamma^*-1)}$ term. Supposing $q_{\mathcal{D}} \leq q_R$ (i.e., there are fewer Ref than RoR calls) and n is small, then $\bar{q}_{\mathsf{H}} \cdot q_R \leq q_R \cdot (q_{\mathsf{H}} + c \cdot q_R)$ for some small constant c. Now if HMAC-DRBG is instantiated at strength $\gamma^* = 256$, it achieves a good security margin up to fairly large q_{H}, q_R. At lower γ^* the margins are less good; however, this is likely an artefact of the proof technique.

Init security. The proof of Init security argues that unless attacker $\mathcal{A}$ queries HMAC on certain points which require guessing the value of either the input I_1 with which HMAC-DRBG was seeded, or an intermediate key / counter computed during setup, then — barring a 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 $\mathsf{M}^{\mathsf{HMAC}}(\varepsilon)$.

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

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

Pres and Rec security. The proofs of Pres and Rec security proceed by bounding: **(1)** the probability that two of the points queried to HMAC during the challenge computation collide; and **(2)** the probability that $\mathcal{A}$ queries HMAC on one of these points. We then argue 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 / collision probabilities of points from the same domain differ throughout the game. This rules out a modular treatment, and complicates the bound. 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 (q_{H}, q_C)-adversary $\mathcal{A}$ in game* Pres *against* HMAC-DRBG *who always outputs addin $\neq \varepsilon$, it holds that*

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

where $\epsilon = q_C \cdot (6q_C + 2n + 8) + 3n + 8$.

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 (q_{H}, q_C)-adversary $\mathcal{A}$ in game* Rec *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*

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

where $\epsilon' = (q_C \cdot (4q_C + 2n + 10 + (q_C - 1) \cdot 2^{-(\gamma^*-1)}) + 3n + 8)$.

This completes our analysis of the Init, Pres, and Rec security of HMAC-DRBG. Combining these results via Theorem 1 then proves Theorem 4.

7 Overlooked Attack Vectors

While the positive results of Sections 5 and 6 are reassuring, the flexibility in the standard to produce variable length and *large* outputs (of up to 2^{19} bits) means that two implementations of the same DRBG may be very different depending on how limits on output production 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 notion of 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 again frustrate a modular treatment). First, 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* $= V$, see Figure 2). Output blocks are produced by iteratively applying a function to the state variables (or in the case of HASH-DRBG, the copy of the state variable). These blocks are concatenated and truncated to β-bits to form the returned output R, and a final state update is performed to produce S'.

Decomposition. We wish to track the evolution of the state variables during a next call relative to the production of different output blocks, in order to reason precisely about the effects of state component compromises at different points. 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 $\mathsf{init} : \text{Seed} \times \mathcal{S} \times \mathbb{N}^{\leq \beta max} \times \{0,1\}^{\leq \alpha} \rightarrow \mathcal{S} \times \{0,1\}^*$ updates the state with additional input prior to output generation, and optionally sets a variable $data \in \{0,1\}^*$. Algorithm $\mathsf{gen} : \text{Seed} \times \mathcal{S} \times \{0,1\}^* \rightarrow \{0,1\}^\ell \times \mathcal{S} \times \{0,1\}^*$ 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 $\mathsf{final} : \text{Seed} \times \mathcal{S} \times \mathbb{N}^{\leq \beta max} \times \{0,1\}^{\leq \alpha} \rightarrow \mathcal{S}$ updates the state post output generation. The next algorithm is constructed from these component parts as shown in the top left panel of Figure 7. The decomposition algorithms $\mathcal{C} = (\mathsf{init}, \mathsf{gen}, \mathsf{final})$ for each of the NIST DRBGs are shown in remaining panels of Figure 7. 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 seed, we omit this parameter.) A diagrammatic depiction of output generation for each of the DRBGs is shown in the full version.

Variable length outputs. Within this iterative structure, the gen subroutine acts like the next algorithm of an internal PRNG, called multiple times within a single next call to produce output blocks. However, as we shall see, the state updates performed by gen do *not* provide forward security after each block [7]. 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 processed during output generation, but who 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, thereby compromising all output in the call. The natural scenario we consider here is a *side channel attack*. Generating multiple output blocks in a single next call results in a significant amount of computation going on 'under the hood' of next — e.g., up to $2^{12} = 4096$ AES-128 computations using a fixed key K^0 for CTR-DRBG with AES-128 — which, given that AES invites leaky implementations [4, 28, 7, 25, 27, 22], is concerning. Since robustness only allows the attacker to compromise the state *after* it has 'properly' updated (via the final process) at the conclusion of a next call, it does not model side channel *during* the call.

Use case: buffering output. As pointed out by Bernstein [5], the overhead incurred by the state update at the conclusion of a CTR-DRBG next call is undesirable. As such, an appealing usage choice is to generate a large output

[7] This is similar to an observation by Bernstein [5] criticizing the inefficiency of CTR-DRBG's update function which appeared concurrently to the production of the first draft of this work. We stress that our modelling of the attack scenario, and systematic treatment of how the issue affects each of the NIST DRBGs, is novel.

```
$\mathsf{next}(X, S, \beta, addin)$
If $cnt > reseed\_interval$
    Return reseed_required
$(S^0, data^0) \leftarrow \mathsf{init}(X, S, \beta, addin)$
$\boxed{\text{If } addin \leftarrow \varepsilon \text{ then } addin \leftarrow 0^n}$
$temp_R \leftarrow \varepsilon\ ; n \leftarrow \lceil \beta/\ell \rceil$
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 $(R, S')$
```

```
HASH-DRBG init
Require: $S = (V, C, cnt), \beta, addin$
Ensure: $S = (V, C, cnt), data$
If $addin \neq \varepsilon$
    $w \leftarrow \mathsf{H}(\mathsf{0x02} \,\|\, V \,\|\, addin)$
    $V \leftarrow (V + w) \bmod 2^L$
$data \leftarrow V$
Return $(V, C, cnt), data$
```

```
HASH-DRBG gen
Require $S = (V, C, cnt), data$
Ensure: $r, S = (V, C, cnt), data$
$r \leftarrow \mathsf{H}(data)$
$data \leftarrow (data + 1) \bmod 2^L$
Return $r, (V, C, cnt), data$
```

```
HASH-DRBG final
Require: $S = (V, C, cnt), \beta, addin$
Ensure: $S = (V, C, cnt)$
$H \leftarrow \mathsf{H}(\mathsf{0x03} \,\|\, V)$
$V \leftarrow (V + H + C + cnt) \bmod 2^L$
$cnt \leftarrow cnt + 1$
Return $(V, C, cnt)$
```

```
HMAC-DRBG init
Require : $S = (K, V, cnt), \beta, addin$
Ensure: $S = (K, V, cnt)$
If $addin \neq \varepsilon$
    $(K, V) \leftarrow \mathsf{update}(addin, K, V)$
Return $(K, V, cnt)$
```

```
HMAC-DRBG gen
Require $(K, V, cnt)$
Ensure $r, S = (K, V, cnt)$
$V \leftarrow \mathsf{HMAC}(K, V)\ ; r \leftarrow V$
Return $r, (K, V, cnt)$
```

```
HMAC-DRBG final
Require : $S = (K, V, cnt), \beta, addin$
Ensure: $S = (K, V, cnt)$
$(K, V) \leftarrow \mathsf{update}(addin, K, V)$
$cnt \leftarrow cnt + 1$
Return $(K, V, cnt)$
```

```
CTR-DRBG init
Require: $S = (K, V, cnt), \beta, addin$
Ensure: $S = (K, V, cnt)$
If $addin \neq \varepsilon$
    If derivation function used then
        $addin \leftarrow \mathsf{CTR\text{-}DRBG\_df}(addin, (\kappa + \ell))$
    Else if $len(addin) < (\kappa + \ell)$ then
        $addin \leftarrow addin \,\|\, 0^{(\kappa + \ell - len(addin))}$
    $(K, V) \leftarrow \mathsf{update}(addin, K, V)$
Return $(K, V, cnt)$
```

```
CTR-DRBG gen
Require: $S = (K, V, cnt)$
Ensure: $r, S = (K, V, cnt)$
$V \leftarrow (V + 1) \bmod 2^\ell\ ; r \leftarrow \mathsf{E}(K, V)$
Return $r, (K, V, cnt)$
```

```
CTR-DRBG final
Require: $S = (K, V, cnt), \beta, addin$
Ensure: $S = (K, V, cnt)$
$(K, V) \leftarrow \mathsf{update}(addin, K, V)$
$cnt \leftarrow cnt + 1$
Return $(K, V, cnt)$
```

Fig. 6: Top left: iterative next algorithm for a DRBG with associated decomposition $\mathcal{C} = (\mathsf{init}, \mathsf{gen}, \mathsf{final})$. Boxed text included for CTR-DRBG only. Right and bottom left: $\mathcal{C} = (\mathsf{init}, \mathsf{gen}, \mathsf{final})$ for HASH-DRBG, HMAC-DRBG and CTR-DRBG.

upfront in a single request, and buffer it to later be used for different purposes [8]. Our attack model investigates the soundness of this 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. 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. As such, our model assumes an attacker learns an output block sent in the clear as e.g., a nonce, in conjunction with the partial state information gleaned via a side

[8] Indeed, NIST SP 800-90A says: “For large generate requests, CTR-DRBG produces outputs at the same speed as the underlying block cipher algorithm encrypts data”, highlighting the efficiency of this approach.

channel. The attacker's goal is to recover *unseen* output blocks used as security critical secrets, thereby breaking the security of the consuming application.

7.1 Attack Model

We now describe our attack model. 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., [26, 15, 1]) introduced significant complexity, 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. We aim to demonstrate key attacks rather than providing an exhaustive treatment.

Attack setup and goals. Consider the next call shown in Figure 7. Letting S denote the state input to next, then this defines a sequence of intermediate states / output blocks passed through 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 $(R, S') = (r^1 \parallel \ldots \parallel r^m, S')$. (For simplicity, we assume the requested number of bits is a multiple of the block length; it is straightforward 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)[9] are known to $\mathcal{A}$. 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.

Extensions. If $addin = \varepsilon$, then init returns the state unchanged, $S^0 = S$. As such, all attacks which succeed when S^0 is partially compromised in our model can also be executed if the relevant component of state S is compromised *prior* to the next call, creating a greater window of opportunity for the attacker.

Security analysis. We analyzed each of the NIST DRBGs with respect to our attack model, and found that each DRBG exhibited vulnerabilities, with CTR-DRBG faring especially badly. We summarize our findings in Figure 8a.

[9] Here we assume the attacker learns a full block and knows its index. This seems reasonable; for example, a TLS client or server random will contain at least one whole block and 12 bytes of a second block (if 4 bytes of timestamp are used). These values would be generated early in a call to the DRBG, and so have a low index j. Both assumptions can be relaxed at the cost of the attacker performing more work to brute-force any missing bits and / or the index.

	(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	✓	✓	×**	×

(a) Table summarizing our analysis. The leftmost three columns correspond to Sections 7.2–7.4. The rightmost column corresponds to Section 7.5. 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.

```
CTR-DRBG//$\mathcal{A}(K^i, r^j, i, j)$
$V^j \leftarrow \mathsf{E}^{-1}(K^i, r^j)$
$V^0 \leftarrow (V^j - j) \bmod 2^\ell$
For $k = 1, \ldots, n$
  $V^k \leftarrow (V^{k-1} + 1) \bmod 2^\ell$
  $r^k \leftarrow \mathsf{E}(K^i, V^k)$
$(K', V') \leftarrow \mathsf{update}(addin, K^i, V^n)$
$cnt' \leftarrow cnt + 1$
$S' \leftarrow (K', V', cnt')$
Return $(\{r^k\}_{k<j}, \{r^k\}_{k>j}, S')$
```

```
HMAC-DRBG//$\mathcal{A}(K^i, r^j, i, j)$
$V^j \leftarrow r^j$
For $k = j + 1, \ldots, n$
  $V^k \leftarrow \mathsf{HMAC}(K^i, V^{k-1})$
  $r^k \leftarrow V^k$
$(K', V') \leftarrow \mathsf{update}(addin, K^i, V^n)$
$cnt' \leftarrow cnt + 1$
$S' \leftarrow (K', V', cnt')$
Return $(\perp, \{r^k\}_{k>j}, S')$
```

```
HASH-DRBG//$\mathcal{A}(data^i, r^j, i, j)$
$data^0 \leftarrow (data^i - i) \bmod 2^L$
For $k = 1, \ldots, n$
  $r^k \leftarrow \mathsf{H}(data^{k-1})$
  $data^k \leftarrow (data^{k-1} + 1)$
Return $(\{r^k\}_{k<j}, \{r^k\}_{k>j}, \perp)$
```

(b) Adversaries for Section 7.2-7.4.

Fig. 7: Summary of analysis (top) and adversaries (bottom) for Section 7.

7.2 CTR-DRBG with Compromised Key

Since each output block encrypts the secret counter V, leakage of the key component of the CTR-DRBG state is especially damaging. Consider attacker $\mathcal{A}$ shown in the left-hand panel of Figure 8b. 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. The key does not update through this process, and so 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 $addin \neq \varepsilon$, $\mathcal{A}$ must guess its value to compute S'.

Discussion. This attack is especially damaging, since target output blocks used as e.g., secret keys will be recovered *irrespective* of their position relative to the block learnt by the attacker, increasing the exploitability of the compromised CTR-DRBG. In comparison, the infamously backdoored DualEC-DRBG only allowed recovery of output produced *after* the compromised block, impacting its practical exploitability [10] (although, of course, the embedded backdoor in DualEC means the attack itself is far easier to execute).

7.3 HMAC-DRBG with Compromised Key

Consider the attacker $\mathcal{A}$ shown in the middle panel of Figure 8b, who compromises the key component K^i 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*).

Past output in a compromised next call. It appears 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. 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 6), 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.

7.4 HASH-DRBG with Compromised Counter.

For HASH-DRBG, it is straightforward to see that if $\mathcal{A}$ learns the counter V^i or its iterating copy $data^i$ for any $i \in [0, n]$, $j \in [1, n]$, then $\mathcal{A}$ achieves goals **(1)** and **(2)** with probability one. 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 8b. However, unlike CTR-DRBG and HMAC-DRBG, without also learning the constant C, achieving goal **(3)** does not seem to be possible in general; we discuss this further in the full version.

7.5 Security of Additional Input

We present an additional attack against $\mathsf{CTR\text{-}DRBG}$ implemented without a derivation function; an appealing implementation choice in terms of efficiency due to the overhead incurred by $\mathsf{CTR\text{-}DRBG\_df}$. Under certain conditions, this allows an attacker who compromises the DRBG state to recover strings of additional input (which may contain secrets) fed to the DRBG during next calls.

The attack. Notice that if $\mathsf{CTR\text{-}DRBG}$ is implemented without a derivation function, then raw strings of additional input are XORed directly into the $\mathsf{CTR\text{-}DRBG}$ state during the application of the update function in next calls (Figure 2, lines 8 and 15). Consider such an implementation of $\mathsf{CTR\text{-}DRBG}$ built from $\mathsf{AES\text{-}128}$. We describe the attack with respect to the 'ideal' conditions. Suppose that attacker $\mathcal{A}$ has compromised the internal state $S = (K, V, cnt)^{10}$, and that the state compromise is followed 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 the full version we describe how to generalize the attack, and discuss how use of the derivation function prevents it.

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

8 Open Source Implementation Analysis

In Section 7, 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 [30] and mbed TLS [8]. We found that between the two libraries these problematic decisions have indeed been made.

Large output requests. Generating many blocks of output in a single request increases 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 number of random bits which may be requested. As such, an arbitrarily large output may be generated using a single key, exacerbating the attacks of Section 7.2. More generally, exceeding the output generation limit increases the success probability 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. The implementation of CTR-DRBG in mbed TLS limits the number of output blocks per next call to 64 blocks of 128-bits; much better for security in the context of our attacks than the 4,096 blocks allowed by the standard. Also, this implementation forces a reseed after 10,000 calls to next; much lower than the allowed maximum of 2^{48}.

Derivation function. In Section 7.5, we described a potential vulnerability in implementations of CTR-DRBG which do not use a derivation function. 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 7.5 may be possible in real world implementations.

Summary. Despite the high level and theoretical nature of our analysis, we found that the problematic implementation decisions which we highlight *are* made in the real world. 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.

9 Conclusion

We conducted an in-depth analysis of NIST SP 800-90A, to investigate unproven security claims and explore 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. 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 Section 6, we saw how the option to call HMAC-DRBG without additional input changed the algorithm in a subtle way which lead to an attack. Similarly, the attacks of Section 7 are both facilitated, and exacerbated, by certain implementation choices allowed by the overly flexible standard. In Section 8, we confirmed that implementers do make these choices in the real world. These may be a warning to standard writers to avoid unnecessary flexibility which may lead to unintended vulnerabilities.

Recommendations. Because these vulnerabilities stem from implementation choices, we can 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. 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. More generally, the design flexibilities we critique above are related to efficiency savings. Designing PRNGs that achieve an optimal balance between security and efficiency is a key direction for future work. 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 Kenny Paterson and the anonymous reviewers for their insightful comments which greatly improved the paper. The first author is supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1); much of this work was completed during an internship at Microsoft Research.

Bibliography

1 Abdalla, M., Belaïd, S., Pointcheval, D., Ruhault, S., Vergnaud, D.: Robust pseudo-random number generators with input secure against side-channel attacks. In: ACNS (2015)

2 Barker, E., Kelsey, J.: NIST SP 800-90A Rev. 1 Recommendation for random number generation using deterministic random bit generators (2015)

3 Barker, E., Kelsey, J.: Draft NIST SP 800-90C. Recommendation for random bit generator (RBG) constructions (2012)

4 Bernstein, D.J.: Cache-timing attacks on AES. Preprint: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf (2005)

5 Bernstein, D.J.: Fast-key-erasure random-number-generators. https://blog.cr.yp.to/20170723-random.html (2017)

6 Bernstein, D.J., Chang, Y.A., Cheng, C.M., Chou, L.P., Heninger, N., Lange, T., Van Someren, N.: Factoring RSA keys from certified smart cards: Coppersmith in the wild. In: ASIACRYPT (2013)

7 Bogdanov, A.: Improved side-channel collision attacks on AES. In: SAC (2007)

8 Butcher, S., Follath, J., García, A.A.: mbed TLS. https://tls.mbed.org/ (2015-2018)

9 Campagna, M.J.: Security bounds for the NIST codebook-based deterministic random bit generator. ePrint (2006)

10 Checkoway, S., Niederhagen, R., Everspaugh, A., Green, M., Lange, T., Ristenpart, T., Bernstein, D.J., Maskiewicz, J., Shacham, H., Fredrikson, M., et al.: On the practical exploitability of Dual EC in TLS implementations. In: USENIX (2014)

11 Cornejo, M., Ruhault, S.: Characterization of real-life PRNGs under partial state corruption. In: ACM CCS (2014)

12 Dodis, Y., Gennaro, R., Håstad, J., Krawczyk, H., Rabin, T.: Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In: CRYPTO (2004)

13 Dodis, Y., Pointcheval, D., Ruhault, S., Vergniaud, D., Wichs, D.: Security analysis of pseudo-random number generators with input:/dev/random is not robust. In: ACM CCS (2013)

14 Dodis, Y., Ristenpart, T., Steinberger, J.P., Tessaro, S.: To hash or not to hash again? (In) differentiability results for H2 and HMAC. In: CRYPTO (2012)

15 Dziembowski, S., Pietrzak, K.: Leakage-resilient cryptography. In: FOCS (2008)

16 FIPS, P.: 140-2. Security Requirements for Cryptographic Modules (2001)

17 Gaži, P., Tessaro, S.: Provably robust sponge-based PRNGs and KDFs. In: EUROCRYPT (2016)

18 Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your ps and qs: Detection of widespread weak keys in network devices. In: USENIX (2012)

19 Hirose, S.: Security analysis of DRBG using HMAC in NIST SP 800-90. In: WISA (2008)

20 Kan, W.: Analysis of underlying assumptions in NIST DRBGs. (2007)

21 Katherine, Q.Y., Green, M., Sanguansin, N., Beringer, L., Petcher, A., Appel, A.W.: Verified correctness and security of mbedTLS HMAC-DRBG. ACM

CCS (2017)
22 Kocher, P., Jaffe, J., Jun, B., Rohatgi, P.: Introduction to differential power analysis. JCEN (2011)
23 Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: CRYPTO (2010)
24 Krawczyk, H., Eronen, P.: Hmac-based extract-and-expand key derivation function (hkdf) (2010)
25 Mangard, S.: A simple power-analysis (SPA) attack on implementations of the AES key expansion. In: International Conference on Information Security and Cryptology (2002)
26 Micali, S., Reyzin, L.: Physically observable cryptography. In: TCC (2004)
27 Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: CT-RSA (2006)
28 Percival, C.: Cache missing for fun and profit (2005)
29 Perlroth, N.: Government announces steps to restore confidence on encryption standards (2013)
30 Project, T.O.: Openssl. https://www.openssl.org/ (1998-2018)
31 Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with composition: Limitations of the indifferentiability framework. In: EUROCRYPT (2011)
32 Ristenpart, T., Yilek, S.: When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In: NDSS (2010)
33 Ruhault, S.: SoK: Security models for pseudo-random number generators. In: IACR Transactions on Symmetric Cryptology (2017)
34 Shrimpton, T., Terashima, R.S.: A provable security analysis of Intel's secure key RNG. In: EUROCRYPT (2015)
35 Shrimpton, T., Terashima, R.S.: Salvaging weak security bounds for blockcipher-based constructions. In: ASIACRYPT (2016)
36 Shumow, D., Ferguson, N.: On the possibility of a back door in the NIST SP800-90 Dual EC PRNG (2007)
37 Turan, M.S., Barker, E., Kelsey, J., McKay, K.A., Baish, M.L., Boyle, M.: SP 800-90B. recommendation for the entropy sources used for random bit generation (2012)
38 Vassilev, A., May, W.: Annex C: Approved random number generators for FIPS PUB 140-2, security requirements for cryptographic modules (2016)
39 Yilek, S., Rescorla, E., Shacham, H., Enright, B., Savage, S.: When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In: ACM SIGCOMM (2009)