An Analysis of the NIST SP 800-90A Standard

Joanne Woodage[1] and Dan Shumow[2]
[1] Royal Holloway, University of London, and [2] Microsoft Research

***Abstract*—We conduct a multi-faceted investigation of the security properties of the three deterministic random bit generator (DRBG) mechanisms recommended in the NIST SP 800-90A standard [4]. This standard received a considerable amount of negative attention, due to the host of controversy and problems with the now retracted DualEC-DRBG, which was included in earlier revisions. 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 provides an analysis of the remaining DRBG algorithms in NIST SP 800-90A. We uncover a mix of positive and less than positive results, emphasizing and addressing the gap between theoretical models, and the NIST DRBGs as specified and used. As an initial positive result, we verify claims in the standard by proving (with a few caveats) the forward security of all three DRBGs. However, digging deeper into flexibility in implementation and usage choices permitted by the standard, we uncover some undesirable properties of these standardized DRBGs. Specifically, we argue that these DRBGs have the property that leaking certain parts of the state may lead to catastrophic failure of the algorithm. Furthermore, we show that flexibility in the specification allows implementers and users of these algorithms to make choices that considerably weaken the algorithms in these scenarios.**

I. Introduction

The NIST Special Publication 800-90A Recommendation for Random Number Generation Using Deterministic Random Bit Generators (NIST SP 800-90A) [4] has had a troubled history. The first version of this publication included the now infamous DualEC-DRBG, which was long suspected to contain a backdoor inserted by the NSA [53]. This suspicion was reportedly later confirmed by documents included in the Snowden leaks [47], leading to a revision of the document that removed the disgraced algorithm.

Perhaps because of the focus on the DualEC-DRBG, the other algorithms standardized in the document have received surprisingly little attention and analysis. These DRBGs — which respectively use a block cipher, a cryptographic hash function, and HMAC as their basic building blocks — are widely used. Indeed, any cryptographic software or hardware seeking FIPS certification *must* implement a pseudorandom generator from the standard. While aspects of the constructions in the SP 800-90A have been analyzed [19], [33], [35], [36], [50]–[52], and certain implementation considerations discussed [16], there has not to date been a deeper analysis of these standardized DRBGs, taking into account the (considerable) flexibility in the algorithm specifications.

The constructions presented in this NIST publication are certainly nonstandard. Even the term Deterministic Random Bit Generator (DRBG) is rare if not totally absent from the literature, which favors Pseudorandom Generator (PRG). Similarly the NIST DRBGs — which return variable (and sizable) length outputs upon request, and support a variety of optional inputs and parameters — do not fit cleanly into the usual security models for PRGs, despite some of these familiar security properties, such as forward security [14], being claimed by the standard. With only a modest amount of formal analysis in the literature to date, coupled with the fact that the standardization of these algorithms did not follow from a competition or widely publicly vetted process, this leaves pseudorandom number generation in large parts of software relying on relatively unanalyzed algorithms.

The goal of this paper is to bridge some of these gaps in analysis in order to highlight areas for improvement, and perhaps motivate a revision to the NIST recommended DRBGs.

Motivation. A secure pseudorandom generator underpins 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 [31], [55], [49], [32], [43], [17], that when a PRG is broken, the security of the reliant application often crumbles with it. Indeed, with much currently deployed cryptography being effectively 'unbreakable' with today's techniques when correctly implemented, exploiting a weakness in the underlying PRG emerges as a highly attractive target for an attacker. As such it is of paramount importance that standardized PRGs are designed to be as secure as possible.

Contributions. In this work, we conduct a thorough investigation into the security of the NIST SP 800-90A DRBGs. We pay particular attention to flexibilities in the specification of these algorithms, which have frequently been abstracted away in previous analysis. On the positive side, we formally prove the forward security of each of the NIST DRBGs, confirming claims made in the standard with concrete bounds. However, 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 forward security model overlooks important attack vectors against these algorithms.

Taking a closer look, we propose an informal security model in which we suppose an attacker compromises part of the state of the DRBG (for example through a side-channel attack) *during* an output generation request. Reconsidered within this framework, we find that each of the constructions admits serious vulnerabilities, and discuss how these weaknesses may be exploited in a TLS handshake. We find a further flaw in a

certain variant of the block-cipher based CTR-DRBG, which allows an attacker who learns a particular part of the state to recover strings of additional input — which may contain secrets — previously fed to the DRBG. We conclude with a number of recommendations for the safe use of these DRBGs based on our findings.

Related work. A number of works have analyzed the pseudorandomness of variants of the NIST DRBGs, in some cases under simplifying assumptions. Campagna [19] proves the pseudorandomness of the CTR-DRBG, under the assumption that it is rekeyed with an independent random state at the commencement of each output generation request. More recently, Shrimpton and Terashima [52] introduce a new security model for block-cipher based constructions, designed to bridge the gap between the standard and ideal cipher models, and obtain bounds on the pseudorandomness of the CTR-DRBG producing a single block of output per request within in. The use of additional input is not modeled in either case. Ruhault discusses the CTR-DRBG derivation function in [50].

Hirose [33] proves the pseudorandomness of the HMAC-DRBG both with and without the use of additional input. Ye et al. in [36] also give a proof of the pseudorandomness of the HMAC-DRBG in the case that additional input is not used, which they use to verify the security and correctness of the mbedTLS implementation of the HMAC-DRBG using the Coq proof assistant.

Work by Kan [35] considers the assumptions underlying the security claims of each of the DRBGs in the standard. Analysis is quite informal and non-standard, and some unusual conclusions drawn.[1] To the best of our knowledge, this is the only work to consider the forward security of the DRBGs; however this is mentioned as an aside and assumed based on the e.g., one-wayness of a cryptographic hash function, as opposed to any formal analysis of the constructions.

II. Preliminaries

Notation. The set of binary strings of length n is denoted by $\{0,1\}^n$. We let $\{0,1\}^*$ denote the set of all binary strings, in which we include the empty string ε. We use $\perp$ to represent the null symbol. We let $x \oplus y$ denote the exclusive-or (XOR) of two strings $x, y \in \{0,1\}^n$, and write $x||y$ to denote the concatenation of two binary strings x and y. We let $\mathrm{left}(x,\beta)$ to denote the leftmost β bits of the 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 $j \in \mathbb{Z}$, 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 $\mathcal{X}$. All logs are to base 2.

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

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

where the superscript denotes a functionality that $\mathcal{A}$ is given oracle access to. In the case that $Dom = Rng$ and F is a permutation, we refer to the above term as the pseudorandom permutation (PRP) distinguishing advantage of $\mathcal{A}$ against F, denoted $\mathsf{Adv}^{\mathrm{prp}}_{\mathsf{F}}(\mathcal{A}, q)$.

A function $\mathsf{E} : \{0,1\}^\kappa \times \{0,1\}^\ell \to \{0,1\}^\ell$ is called a block-cipher if for each $K \in \{0,1\}^\kappa$, $\mathsf{E}(K,\cdot)$ is a permutation on $\{0,1\}^\ell$. We let $\mathsf{D}(K,\cdot)$ denote the inverse, or decryption, under K, so $\mathsf{D}(K, \mathsf{E}(K,m)) = m$ for all $m \in \{0,1\}^\ell$.

Pseudorandom number generators. A pseudorandom number generator (PRG) is used to convert a small amount of statistical entropy into pseudorandom bits strings of arbitrary (bounded) length. Formally, we define a PRG to be a pair of deterministic algorithms $\mathsf{PRG} = (\mathsf{setup}, \mathsf{generate})$. The algorithm $\mathsf{setup} : \{0,1\}^* \to \mathcal{S}$ takes as input a string $I \in \{0,1\}^*$ and returns an initial state $S_0 \in \mathcal{S}$, where $\mathcal{S}$ denotes the state space of the PRG. Looking ahead, we will assume that I has a degree of *statistical entropy*. The algorithm $\mathsf{generate} : \mathcal{S} \times \mathbb{N} \times \{0,1\}^* \to (\{0,1\}^\beta \cup \{\perp\}) \times \mathcal{S}$ takes as input a state $S \in \mathcal{S}$, the requested number of bits β, and an optional string of additional input $addin \in \{0,1\}^*$, and returns an output $R \in (\{0,1\}^\beta \cup \{\perp\})$, along with an updated state $S' \in \mathcal{S}$.[2] The SP 800-90A standard uses the term *deterministic random bit generator* (DRBG) instead of the more familiar PRG. We shall use both terms interchangeably throughout the document.

III. The NIST SP 800-90A Standard

In this section, we give an overview of the NIST SP 800-90A standard. The standard defines three DRBG mechanisms, the HASH-DRBG, HMAC-DRBG, and CTR-DRBG, each based on a different primitive — namely a strong cryptographic hash function, HMAC and an approved block-cipher respectively. These algorithms are designed to produce pseudorandom output of varying length upon request. For the instantiations described here (and for all those permitted in the standard, with the exception of the CTR-DRBG instantiated with 3-KeyTDEA [8]), up to 2^{19} bits of output may be requested in each generate call.

Entropy sources. The DRBGs must have access to an *approved entropy source* during instantiation (that is to say, a live entropy source as approved in [5] or a (truly) random bit generator as per [6]) in order to generate its initial state S_0. The

[1] For example, concluding that the CTR-DRBG is flawed and unsafe to use, since a block-cipher in counter mode will never produce a repeated output block.

[2] We note that here we have extended the usual PRG definition to better capture the NIST generators, by allowing variable length outputs to be requested, and additional input to be incorporated during output generation. Following Shrimpton et al. in [51], we also allow the generate algorithm to return $\perp$ to represent it blocking in response to an invalid request.

DRBG is initially seeded with entropy equal to the *security strength* of the instantiation, a parameter measuring the amount of work required to 'break' the DRBG, and which is equal to the key size of the underlying block-cipher, or the output length of the hash function, for the constructions described here. If the DRBG has continual access to an approved entropy source, then it may support *prediction resistance*, in which case entropy inputs are periodically incorporated into the state via the reseed algorithm. Reseeds may be explicitly requested by the consuming application, triggered by a request for *prediction resistance* in a generate call, or will be forced every 2^{48} consecutive output generation requests where a state component cnt is used to keep track of the number of generate calls since the last reseed. We do not model reseeding in this work, and so omit parameters indicating whether prediction resistance is supported from the state.

Derivation functions. Both the CTR-DRBG and the HASH-DRBG use a *derivation function* to condition entropy inputs and (and in the case of the CTR-DRBG, additional input strings), prior to their incorporation into the generator state. Use of the derivation function for the CTR-DRBG in particular represents a significant computational overhead; therefore if the DRBG is implemented with a *full entropy source* — that is to say a source which returns uniform bits strings as opposed to just those with high entropy — then the standard offers a second variant of the CTR-DRBG which does not use a derivation function. This represents an "implementation tradeoff" [7], since not using a derivation function makes for a much more efficient implementation, however the instantiation is then restricted to be implemented with a full entropy source, and can only use additional input strings of at most $\kappa + \ell$-bits in length, where κ, ℓ denote the key and block size of the block-cipher respectively. We discuss both variants in Section III-A.

Additional Input. In addition to inputs drawn from the entropy source, the NIST SP 800-90A standard gives the option for strings of additional input (denoted $addin$), to be fed into the state of the DRBG during generate and reseed calls. The standard permits these strings to be public information (e.g., device serial numbers and time stamps), or may contain secrets provided they do not require protection at a higher security strength than the DRBG[3]. Unlike inputs drawn from the entropy source, these inputs are not required to contain entropy and the standard emphasizes that they are optional. That said, if these inputs do contain entropy, then their use may provide a buffer in the event of a system failure or compromise.

Validity checks. For simplicity, we have abstracted a number of checks performed at the commencement of a generate call — such as confirming the reseed counter has not been exceeded (which would force a reseed prior to output production), and blocking if the number of bits requested exceeds the maximum allowed length — into a procedure check : $\mathcal{S} \times \mathbb{N} \times \{0,1\}^* \to \{b\}$ which checks the validity of the input tuple $(S, \beta, addin)$, and outputs 1 only if these checks are passed.

Iterative generate **algorithms.** The generate algorithm of each of the NIST DRBGs has the same high-level structure. In order to track the evolution of the DRBG state during a single output generation request, it shall be useful to formalize this structure, and introduce some extra notation to this end here.

We say that a DRBG has an *iterative* generate *algorithm* if the generate algorithm may be decomposed into a tuple of subroutines $\mathcal{C} = (\mathsf{init}, \mathsf{next}, \mathsf{final})$, where init / final : $\mathcal{S} \times \mathbb{N} \times \{0,1\}^* \to \mathcal{S}$ update the state prior / post output generation, and next : $\mathcal{S} \to \{0,1\}^\ell \times \mathcal{S}$ takes as input a given state S and returns an output block r of some fixed length ℓ and an updated state S'.

The generate algorithm is constructed from these subprocedures as follows. When generate is called on some input $(S, \beta, addin)$, this tuple is first submitted to the check algorithm which tests whether the input passes any validity checks which may be performed by the DRBG. If the input passes, init updates the state with the additional input string $S^0 \leftarrow \mathsf{init}(S, \beta, addin)$, to which next is iteratively applied to compute $(r^j, S^j) \leftarrow \mathsf{next}(S^{j-1})$ for $j = 1, \ldots m$ where $m = \lceil \beta/\ell \rceil$. The final state S^m is then updated via the final procedure $S' \leftarrow \mathsf{final}(S^m, \beta, addin)$, and the output of generate is set to (R, S') where $R = \mathrm{left}(r_1 || \ldots || r_m, \beta)$. This decomposition surfaces a distinction between the internal state updates following the production of each block of output, and the final state update executed by final at the conclusion of the generate call.

A diagram showing the evolution of the state within a generate call in terms of these subroutines is given in Figure 1. Looking ahead, when we wish to consider the effect of state leakage 'within a generate call', we mean full or partial leakage of one of the states $S^0, \ldots, S^m$ passed through during the iterative output generation process.

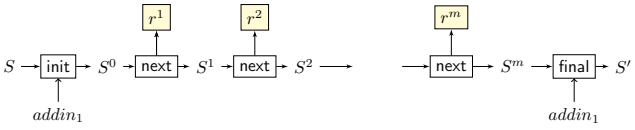

Fig. 1: Diagram showing the production of output block for a DRBG with an iterative generate algorithm, with associated decomposition $\mathcal{C} = (\mathsf{init}, \mathsf{next}, \mathsf{final})$.

With this in place, we now describe the generate algorithms of each of the DRBGs. We will assume that each DRBG is initialized with an 'ideal' state, as opposed to one derived from the entropy source, and so omit the setup algorithms here. The same simplifying assumption is made in all other analyses of these DRBGs that we are aware of; extending analysis to take into account the process by which each of the DRBGs derives their initial state from the entropy source

[3]To quote directly from the standard for clarity, "if the additional input contains secret / private information (e.g., a social security number), that information shall not require protection at a higher security strength than the security strength supported by the DRBG." [7].

is an important direction for future work. For completeness, we include the setup algorithms in Appendix C. We also give examples of typical instantiations of the DRBGs; the corresponding parameter settings are shown in Figure 6 in Appendix A. The full list of allowed instantiations is given in the SP 800-90A standard [7].

A. *The* CTR-DRBG *DRBG*

The CTR-DRBG is based upon a block cipher $\mathsf{E} : \{0,1\}^\kappa \times \{0,1\}^\ell \to \{0,1\}^\ell$ used in CTR-mode. The state is of the form $S = (K, V, cnt)$ where $K \in \{0,1\}^\kappa$ is used as a key for the block cipher, $V \in \{0,1\}^\ell$ is the counter, and cnt denotes the reseed counter. The standard states that K and V are the security critical state variables. We assume setup returns an initial state $S_0 = (K_0, V_0, cnt_0)$ where $cnt_0 = 1$ and $K_0 \leftarrow\!\!\$\ \{0,1\}^\kappa$, $V_0 \leftarrow\!\!\$\ \{0,1\}^\ell$. We present the generate algorithm for the CTR-DRBG below. For concreteness, one may assume the CTR-DRBG is instantiated with AES-128 [48].

Algorithms. The update algorithm is used to update the state variables K and V in a one-way fashion, and incorporate any additional input (which is passed to the update algorithm via the parameter $provided\_data$). The algorithm (shown below) runs the block cipher in CTR-mode using the current key and counter, concatenating the resulting output blocks. These are then truncated to the leftmost $\kappa + \ell$ bits and the bit-string $provided\_data$ is XORed in. Looking ahead, the way in which the update function is called by each of the other processes ensures the provided data is always exactly $\kappa+\ell$ bits in length.

Algorithm 1 CTR-DRBG update

Require: : $provided\_data, K, V$
Ensure: : K, V
  $temp \leftarrow \varepsilon$
  $m \leftarrow \lceil (\kappa + \ell)/\ell \rceil$
  for $j = 1, \ldots, m$ **do**
    $V \leftarrow (V + 1) \bmod 2^\ell$
    $C_i \leftarrow \mathsf{E}(K, V)$
    $temp \leftarrow temp || C_i$
  $temp \leftarrow \text{left}(temp, (\kappa + \ell))$
  $K||V \leftarrow temp \oplus provided\_data$
  return (K, V)

As discussed earlier in the section, there are two variants of the CTR-DRBG depending on whether a derivation function is used. The derivation function CTR-DRBG df combines a CBC-MAC-based conditioning function with a mixing step, with the aim of extracting a near uniform output from sufficiently high-entropy inputs. A full pseudocode description is given in Appendix C. In the case that the derivation function is not used, the strings of additional input $addin$ are restricted to be at most $\kappa + \ell$-bits in length.

The init subroutine (lines 3 - 10) incorporates any additional input via the update function — if a derivation function is used, the string of additional input is conditioned into a string of $(\kappa + \ell)$-bits with the CTR-DRBG df prior to this process. If additional input is not used, the state is left unchanged and $addin$ is set to $0^{(\kappa+\ell)}$ (this is to form an input to update during the final procedure at the conclusion of the call). Output blocks are then iteratively generated using the block cipher in CTR-mode (each block / counter increment corresponding to an iteration of the next subroutine in lines 13 - 14). The key K remains unchanged throughout these iterations. At the conclusion of the call, the final process updates both K and V via an application of the update function (line 17). A diagrammatic depiction of the generate algorithm for the CTR-DRBG is given in Figure 2.

Algorithm 2 CTR-DRBG generate

Require: : $S = (K, V, cnt), \beta, addin$
Ensure: : $S' = (K', V', cnt'), R$
1: **if** $0 \leftarrow \mathsf{check}(S, \beta, addin)$ **then**
2: Return (error, $\perp$)
3: **if** $addin \neq \varepsilon$ **then**
4: **if** derivation function used **then**
5: $addin \leftarrow \mathsf{df}(addin, (\kappa + \ell))$
6: **else if** $len(addin) < (\kappa + \ell)$ **then**
7: $addin \leftarrow addin || 0^{(\kappa+\ell-len(addin))}$
8: $(K^0, V^0) \leftarrow \mathsf{update}(addin, K, V)$
9: **else**
10: $addin \leftarrow 0^{\kappa+\ell}$; $(K^0, V^0) \leftarrow (K, V)$
11: $temp \leftarrow \varepsilon$; $m \leftarrow \lceil \beta/\ell \rceil$
12: **for** $j = 1, \ldots, m$ **do**
13: $V^j \leftarrow (V^{j-1} + 1) \bmod 2^\ell$
14: $r^j \leftarrow \mathsf{E}(K^0, V^j)$
15: $temp \leftarrow temp || r^j$
16: $R \leftarrow \text{left}(temp, \beta)$
17: $(K', V') \leftarrow \mathsf{update}(addin, K^0, V^m)$
18: $cnt' \leftarrow cnt + 1$
19: **return** $(R, (K', V', cnt'))$

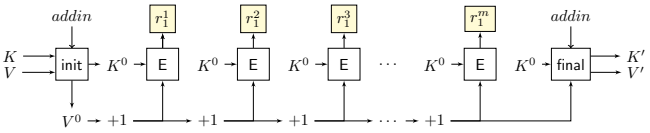

Fig. 2: Evolution of state of the CTR-DRBG within a single generate call, with initial state $S = (K, V, cnt)$.

B. *The* HMAC-DRBG

HMAC is a keyed-hash message authentication code, which was introduced by Bellare et al. in [10] and subsequently standardized [54]. The HMAC-DRBG uses $\mathsf{HMAC} : \{0,1\}^\ell \times \{0,1\}^* \to \{0,1\}^\ell$ to generate blocks of pseudorandom output. The state is of the form $S = (K, V, cnt)$ where the standard defines K and V to be the security critical secret state variables. We assume that setup returns an initial state $S_0 = (K_0, V_0, cnt_0)$ where $cnt_0 = 1$ and $K_0, V_0 \leftarrow\!\!\$\ \{0,1\}^\ell$. Here $K \in \{0,1\}^\ell$ is used as the HMAC key, $V \in \{0,1\}^\ell$ is a counter, and cnt denotes the reseed counter. For concreteness, one may assume the HMAC-DRBG is instantiated with HMAC / SHA-256.

Algorithms. The generate algorithm makes use of a subroutine named update, which is used to incorporate any additional input into the state variables K and V, and update both in a one-way manner. Notice that if additional input is included in the call, an extra pair of updates is executed.

Algorithm 3 HMAC-DRBG update

Require: : $addin, K, V,$
Ensure: : K, V
  $K \leftarrow \mathsf{HMAC}(K, V||0\text{x}00||addin)$
  $V \leftarrow \mathsf{HMAC}(K, V)$
  if $addin \neq \varepsilon$ **then**
    $K \leftarrow \mathsf{HMAC}(K, V||0\text{x}01||addin)$
    $V \leftarrow \mathsf{HMAC}(K, V)$
  return (K, V)

The generate algorithm for the HMAC-DRBG proceeds as follows. The init process (lines 3 - 5) incorporates any additional input into the state variables via the update function; if additional input is not included in the call, the state is left unchanged. Letting K^0, V^0 denote the state variables following this process, output blocks are then generated by iteratively applying $\mathsf{HMAC}(K^0, \cdot)$ to the current counter V^{j-1}, and setting both the next output block r^j and the next counter value V^j equal to the the resulting string (this constitutes the next subroutine, depicted in lines 8 - 9). The key K^0 remains unchanged through each iteration. At the conclusion of the call, the final process updates both K and V via the update function (line 12). A diagrammatic depiction of the generate algorithm for the HMAC-DRBG is given in Figure 3.

Algorithm 4 HMAC-DRBG generate

Require: $S = (K, V, cnt)$, $\beta, addin$
Ensure: $S' = (K', V', cnt'), R$
1: **if** $0 \leftarrow \mathsf{check}(S, \beta, addin)$ **then**
2: **return** $(\mathsf{error}, \perp)$
3: **if** $addin \neq \varepsilon$ **then**
4: $(K^0, V^0) \leftarrow \mathsf{update}(addin, K, V)$
5: **else** $(K^0, V^0) \leftarrow (K, V)$
6: $temp \leftarrow \varepsilon$; $m \leftarrow \lceil \beta/\ell \rceil$
7: **for** $j = 1, \ldots, m$ **do**
8: $V^j \leftarrow \mathsf{HMAC}(K^0, V^{j-1})$
9: $r^j \leftarrow V^j$
10: $temp \leftarrow temp||r^j$
11: $R \leftarrow \mathsf{left}(temp, \beta)$
12: $(K', V') \leftarrow \mathsf{update}(addin, K^0, V^m)$
13: $cnt' \leftarrow cnt + 1$
14: **return** $(R, (K', V', cnt'))$

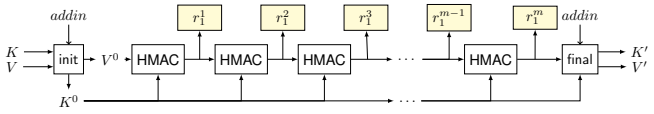

Fig. 3: Evolution of state of the HMAC-DRBG within a single generate call, with initial state $S = (K, V, cnt)$.

C. *The* HASH-DRBG *DRBG*

The HASH-DRBG is based on an (unkeyed) cryptographic hash function $\mathsf{SH} : \{0,1\}^* \rightarrow \{0,1\}^\ell$; here we assume that the HASH-DRBG is instantiated with SHA-256. The state is of the form $S = (V, C, cnt)$, where $V \in \{0,1\}^{len}$ is a counter which is hashed to produce output blocks, and cnt again denotes the reseed counter. We assume the setup algorithm returns an initial state $S_0 = (V_0, C, cnt_0)$ where $cnt_0 = 1$, $V \leftarrow_\$ \{0,1\}^{len}$, and C is deterministically derived from V_0 using the HASH-DRBG derivation function. A pseudocode description of this derivation function is given in Appendix C.

The constant C is only updated during a reseed (when it is again derived from the new V variable), and is added into the state variable V during each state update. The standard does not explicitly state the purpose of C; however slides from 2004 and written by Kelsey [38] on the NIST DRBGs describe how they "Hash with constant to avoid duplicating other hash computations"; thus it would appear its purpose is to ensure that even if a previous counter V is duplicated at some point in a different reseed period, the inclusion of the (almost certainly distinct) counter in the subsequent state update prevents the previous sequence of states being repeated. The standard defines V and C to be the security critical state variables.

Algorithms. We present the generate algorithm for the HASH-DRBG below. If additional input is used in the generate call, it is hashed and added into the counter V modulo 2^{len} during the init process depicted in lines 3 - 5. Output blocks are then produced by hashing the counter V in CTR-mode (constituting iterations of the next process given in lines 10 - 11). At the conclusion of the call, the final routine (lines 14 - 15) hashes the counter with a distinct prefix prepended, and the resulting string — along with the constant C and cnt — are added to V, the result of which is set as the updated counter. Notice the domain separation induced by prepending a distinct byte to the input to SH in the different stages of the algorithm. A diagrammatic depiction of the generate algorithm for the HASH-DRBG is given in Figure 4.

Algorithm 5 HASH-DRBG generate

Require: : $S = (V, C, cnt), \beta, addin$
Ensure: : $S' = (V', C, cnt'), R$
1: **if** $0 \leftarrow \mathsf{check}(S, \beta, addin)$ **then**
2: Return $(\mathsf{error}, \perp)$
3: **if** $addin \neq \varepsilon$ **then**
4: $w \leftarrow \mathsf{SH}(0\text{x}02||V||addin)$
5: $V^0 = (V + w) \bmod 2^{len}$
6: **else** $V^0 \leftarrow V$
7: $temp \leftarrow \varepsilon$
8: $m \leftarrow \lceil \beta/\ell \rceil$
9: **for** $j = 1, \ldots, m$ **do**
10: $r^j \leftarrow \mathsf{SH}(V^{j-1})$
11: $V^j \leftarrow (V^{j-1} + 1) \bmod 2^{len}$
12: $temp \leftarrow temp||r^j$
13: $R \leftarrow \mathsf{left}(temp, \beta)$
14: $H \leftarrow \mathsf{SH}(0\text{x}03||V^0)$
15: $V' \leftarrow (V^0 + H + C + cnt) \bmod 2^{len}$
16: $cnt' \leftarrow cnt + 1$
17: **return**(V', C, cnt')

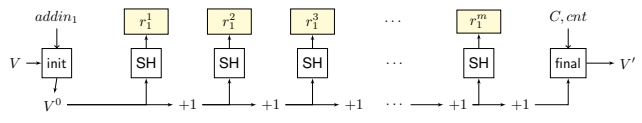

Fig. 4: Evolution of state of the HASH-DRBG within a single generate call, with initial state $S = (V, C, cnt)$.

IV. Security Analysis in Conventional Models

Now that we have described the specification of the NIST DRBGs, we turn our attention to their security properties.

Security claims. The standard states that each of the DRBG

mechanisms is 'backtracking resistant'. In the case that the DRBG is implemented with access to a live entropy source, it is also claimed to be 'prediction resistant'.

The former security property is defined in the standard as the familiar forward security notion for DRBGs, first formalized by Bellare et al. in [14], which guarantees that if an attacker compromises the state of the DRBG at some point in time, then all output bits produced prior to the point of compromise still appear pseudorandom. The latter property ensures that if the state of the DRBG is compromised, and then refreshed with sufficient entropy, then future output will again appear pseudorandom. Somewhat surprisingly, to the best of our knowledge, neither of these properties have been formally investigated and proved for the NIST DRBGs.

In this section, we address this gap and prove the forward security of each of the NIST DRBGs under standard assumptions about the underlying primitives.

Forward security / backtracking resistance. The definition of forward security / backtracking resistance is defined somewhat informally in the standard via a diagram and written discussion; for concreteness we recall the formal definition below, which we extend to better capture the NIST DRBGs by explicitly modelling the use of additional input strings.

The forward security game is depicted in Figure 5, where the forward security advantage of an attacker $\mathcal{A}$ in the game is defined

$$\mathrm{Adv}^{fwd}_{\mathsf{DRBG},\beta}(\mathcal{A},q) = 2\left|\Pr\left[\,\mathrm{Fwd}^{\mathcal{A},q}_{\mathsf{DRBG},\beta} \Rightarrow 1\,\right] - \frac{1}{2}\right| .$$

As per the usual forward security definition, we consider a DRBG responding to a series of q output generation requests of β bits[4]. Since we do not wish to model reseeds here, we assume $q < 2^{48}$, the maximum number of output requests permitted before a reseed is forced; the size of this number means that this restriction is of minimal practical concern. Since we are primarily interested in the security of generated output, we will assume that all input tuples are valid.

We extend the usual forward security definition to allow these calls to incorporate additional input strings $addin$. The case in which no additional input is captured by setting $addin_i = \varepsilon$ for $i = 1, \ldots, q$. We make no assumptions on the entropy or structure of the additional input (other than that it does not exceed the maximum allowed length), and give all additional input strings to the attacker in the guessing phase of the game; this is a conservative assumption since any degree of entropy in the additional input strings can only make the attacker's job harder.

With our security model in place, we now analyze the forward security of each of the NIST DRBGs. We provide general security bounds making no assumption on the choice of underlying primitive.

[4]Our proofs may easily be extended to the case in which a different number of output bits are requested in each call.

A. Forward security of the CTR-DRBG.

We begin by proving the forward security of the CTR-DRBG, via a reduction to the PRP-security of the underlying block-cipher.

Theorem 4.1: Let the CTR-DRBG be as described in Section III-A instantiated with a block-cipher $\mathsf{E} : \{0,1\}^\kappa \times \{0,1\}^\ell \rightarrow \{0,1\}^\ell$, where we assume setup returns state $S_0 = (K_0, V_0, cnt_0)$ with $K_0, V_0 \leftarrow\!\!\$\ \{0,1\}^\ell$. Suppose that additional input is used in every call. Let $\mathcal{A}$ be an attacker in game $\mathrm{Fwd}^{\mathcal{A},q}_{\mathsf{CTR\text{-}DRBG},\beta}$ against the CTR-DRBG running in time T. Then there exist adversaries $\mathcal{B}$, $\mathcal{C}$ such that

$$\begin{aligned}\mathsf{Adv}^{\mathrm{fwd}}_{\mathsf{CTR\text{-}DRBG},\beta}(\mathcal{A},q) = 3q \cdot \mathsf{Adv}^{\mathrm{prp}}_{\mathsf{E}}(\mathcal{B},m_1) \\ + q \cdot \mathsf{Adv}^{\mathrm{prp}}_{\mathsf{E}}(\mathcal{C}, m_1+m_2) \\ + \frac{q\cdot(3m_1^2 + (m_1+m_2)^2)}{2^{\ell+1}} .\end{aligned}$$

With the above conditions the same, except that additional input is not used, then for any attacker $\mathcal{A}$ in the game $\mathrm{Fwd}^{\mathcal{A},q}_{\mathsf{CTR\text{-}DRBG},\beta}$, there exists an adversary $\mathcal{B}$ such that

$$\begin{aligned}\mathsf{Adv}^{\mathrm{fwd}}_{\mathsf{CTR\text{-}DRBG},\beta}(\mathcal{A},q) = q \cdot \mathsf{Adv}^{\mathrm{prp}}_{\mathsf{E}}(\mathcal{B},m_1) + \\ q \cdot \mathsf{Adv}^{\mathrm{prp}}_{\mathsf{E}}(\mathcal{C}, m_1+m_2) \\ + \frac{q\cdot(m_1^2 + (m_1+m_2)^2)}{2^{\ell+1}} .\end{aligned}$$

In both cases $m_1 = \lceil \frac{\kappa+\ell}{\ell} \rceil$, $m_2 = \lceil \frac{\beta}{\ell} \rceil$, and $\mathcal{B}$ and $\mathcal{C}$ run in time $T' \approx T$.

The proof follows from a standard hybrid argument, showing that we can sequentially interchange block-cipher E outputs with random bit strings, via a reduction to first the PRP-security of E and then applying the PRP/PRF-switching lemma (see e.g., [13] for a proof of this result). During state updates, any additional input is XORed into these (now random) keys / counters; therefore the updated states are uniform too, regardless of the content of the additional input. The full proof is given in Appendix A1. The difference between the two bounds is due to the the additional m_1 block-cipher calls made at the beginning of each generate call if additional input is used. Since $m_1 = \lceil \frac{\kappa+\ell}{\ell} \rceil$ is small (for example, $m_1 = 2$ in the case that the CTR-DRBG is instantiated with AES-128), this corresponds to only a small increase in the advantage term.

B. Forward security of the HMAC-DRBG.

In this section, we prove the forward security of the HMAC-DRBG via a reduction to the PRF security of HMAC. Bellare [9] proved that HMAC is a PRF under the assumption that the underlying compression function is a PRF via a non-uniform reduction; a uniform proof of this result was later given by Gazi et al. in [30].

A caveat when additional input is not used. An interesting observation is that — contrary to claims made in the standard — the HMAC-DRBG is not *strictly* forward-secure in the usual sense if additional input is not used. This is because the final state $S_q = (K_q, V_q, cnt_q)$ is such that $V_q = \mathsf{HMAC}(K_q, r_q^m)$ where r_q^m denotes the final output block produced in the

q^{th} generate call. As such the attacker can easily check whether this block is real or random given the state S_q. This observation is implicit in the proof of the pseudorandomness of the HMAC-DRBG by Hirose [33], although the connection to forward security is not made in this work.

That said, we can still prove that all output produced by the HMAC-DRBG *apart* from the last block of the q^{th} generate call remain pseudorandom conditioned on knowledge of the state S_q. We do this by defining game $\text{Fwd}^{*\mathcal{A},q}_{\mathsf{DRBG},\beta}$ to be identical to game $\text{Fwd}^{*\mathcal{A},q}_{\mathsf{DRBG},\beta}$ except the attacker is given $R^*_q = r^1_q||\ldots||r^{m-1}_q$ in his challenge if $b = 0$ (and a random bit string of the same length of $b = 1$), as opposed to the full output block $R_q = \text{left}(r^1_q||\ldots||r^m_q, \beta)$.

Theorem 4.2: Let the HMAC-DRBG be as described in Section III-B, where we assume setup returns state $S_0 = (K_0, V_0, cnt_0)$ with $K_0, V_0 \leftarrow\!\!\$\ \{0,1\}^\ell$. Suppose that additional input is used in every call. Let $\mathcal{A}$ be an attacker in game $\text{Fwd}^{\mathcal{A},q}_{\mathsf{HMAC\text{-}DRBG},\beta}$ against the HMAC-DRBG running in time T. Then there exist adversaries $\mathcal{B}$, $\mathcal{C}$ such that

$$\mathsf{Adv}^{\text{fwd}}_{\mathsf{HMAC\text{-}DRBG},\beta}(\mathcal{A},q) \le 6q \cdot \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{B},2) + 2q \cdot \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{C},m+2) + \frac{q\cdot(m+1)^2}{2^\ell}\ .$$

With the above conditions the same, except that additional input is not used, then for any attacker $\mathcal{A}$ in the game $\text{Fwd}^{*\mathcal{A},q}_{\mathsf{HMAC\text{-}DRBG},\beta}$, there exists an adversary $\mathcal{B}$ such that

$$\mathsf{Adv}^{\text{fwd}^*}_{\mathsf{HMAC\text{-}DRBG},\beta}(\mathcal{A},q) \le 2q \cdot \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{B},m+1) + \frac{q\cdot m^2}{2^\ell}\ .$$

In both cases $m = \lceil \beta/\ell \rceil$, and $\mathcal{B}$ and $\mathcal{C}$ run in time $T' \approx T$.

The proof follows from a hybrid argument, showing that we can sequentially interchange HMAC outputs with random bits strings via a reduction to the PRF-security of HMAC. The first half of the proof replaces output and state values with random bit strings, and is similar to the proof of pseudorandomness by Hirose in [33]. We then argue that we can reverse these steps for the state computations to return to the real final state, while preserving the truly random outputs. The full proof is given in Appendix B1. The difference between the two bounds is due to the additional HMAC applications (under rotating keys) that are required to incorporate additional input into the state at the beginning and conclusion of each generate call.

C. Forward security of the HASH-DRBG.

In this section, we prove the forward security of the HASH-DRBG under the assumption that the underlying hash function SH is a *random oracle* [12], a standard heuristic approach to modelling the security properties of cryptographic hash function. Accordingly, the attacker $\mathcal{A}$ in the forward security game is given access to the oracle SH, and we measure his resources in the number of queries made to the SH. We provide a proof for the case in which additional input is not used here; the case in which it is used is likely similar.

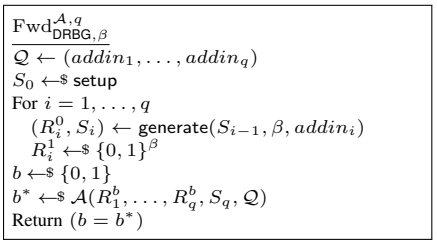

Fig. 5: The forward security game for a DRBG DRBG.

Theorem 4.3: Let the HASH-DRBG be as shown in Section III-C, instantiated with a random oracle SH : $\{0,1\}^* \to \{0,1\}^\ell$, where we assume setup returns the initial state $S_0 = (V_0, C, cnt_0)$ with $V_0 \leftarrow\!\!\$\ \{0,1\}^{len}$, and C is derived from V_0 as per the specification of the algorithm. We assume that $\ell < len - 48$. Suppose that additional input is not used. Let $\mathcal{A}$ be an attacker in game $\text{Fwd}^{\mathcal{A},q}_{\mathsf{HASH\text{-}DRBG},\beta}$ against the HASH-DRBG who makes at most queries σ to the random oracle SH. Then

$$\mathsf{Adv}^{\text{fwd}}_{\mathsf{HASH\text{-}DRBG},\beta}(\mathcal{A},q) \le \frac{q(\sigma(m+1)+3(q+1)(m-1))}{2^\ell}\ ,$$

where $m = \lceil \beta/\ell \rceil$.

While it may seem obvious that, under the assumption that the underlying hash function is a random oracle, output produced by the HASH-DRBG is indistinguishable from random, actually bounding the attack success probability is surprisingly fiddly. The proof proceeds by removing collisions in the inputs to the random oracle SH, allowing us to replace them with uniform bit strings chosen independently of the inputs, then arguing that the attacker cannot distinguish this change unless he can guess one of the previous state values which constitute the inputs to the random oracle. We assume in the theorem that $\ell < len - 48$; this holds for all instantiations permitted by the standard. The full proof is given in Appendix B2.

V. Overlooked Attack Vectors

The positive results on the forward-security of the NIST DRBGs in Section IV certainly offer reassurance on the security of previously generated output in the event of state compromise. However in this section, we argue that by applying the standard forward-security definition to the (fairly non-standard) NIST DRBGs without adaptation overlooks important attack vectors against this particular set of DRBGs, when they are used in certain ways permitted by the standard.

We emphasize that the forward-security definition used in this work is set as a goal (albeit stated informally) in the standard, so this dissonance between model and construction is not a result of analyzing the security of the constructions within a framework for which they were not intended. Likewise, the points made in this section do not contradict the security bounds made in the previous section. Rather we argue that in certain (realistic) scenarios — namely when the DRBG is used to produce many output blocks per generate call —

it is worth taking a closer look at all the points during output production at which a state may be compromised.

Variable length outputs. Stateful pseudorandom number generators in the literature, are typically defined to produce a single fixed length output block per request, that is to say each output generation request returns $(r, S') \leftarrow \mathsf{generate}(S)$ where $r \in \{0,1\}^\ell$ (see, for example, [2], [14], [24]–[26] for examples of this definition in a variety of different contexts). There is no option to request variable length outputs; therefore if more output blocks are required than can be returned in one generate call, the consuming application must make multiple output generation requests, where each such request triggers a 'proper' state update.

In contrast, the NIST DRBGs allow for variable and large amounts of outputs— up to 2^{19} bits— to be requested in each generate call. This corresponds to many blocks of output being produced to satisfy a single request. The generate algorithm produces these output blocks by repeatedly calling the next subroutine to produce $(r^j, S^j) \leftarrow \mathsf{next}(S^{j-1})$ for $j = 1, \ldots, \lceil \beta/m \rceil$ and returning $R_j = \text{left}(r^1 || \ldots || r^m, \beta)$. As such the next subroutine effectively acts as an internal PRG (defined in the usual, one block of output per request sense), which is called multiple times within a single generate call. As we shall see, the state updates performed by the next subroutine do not provide forward security after each block.

Side channels and partial state leakage. The result of this is that when multiple output blocks are generated per generate request, there is a significant amount of active computation going on 'under the hood' of the generate algorithm. For a concrete example, using the CTR-DRBG with AES-128 to generate the maximum 2^{19} bits of output in a given request corresponds to up to $2^{12} = 4096$ AES-128 computations using a single key K in *each* generate call. Given that it is well known that AES invites leaky implementations [15], [18], [41], [44]–[46], to assume that the attacker can never learn even partial state information about the key used during these computations is rather optimistic.

Comparison to existing security models. The standard forward-security definition for DRBGs shown in Figure 5 only allows the attacker to compromise the state *after* it has been 'properly' updated by the final process at the conclusion of a generate call. Other security models which model state compromise on pseudorandom number generators with input such as the various notions of *robustness* [3], [27], [23], [29], [1] for DRBGs implemented with a continual access to an entropy source, allow the attacker to compromise any of the states $S_0, \ldots, S_q$ (although security is not then expected until the state has been refreshed with sufficient entropy). However there is no means for an attacker to learn any of the intermediate states passed through within a single generate call. .

This is not to criticize these security models — which were designed with DRBGs producing a single block of output per request in mind, in which case this is an entirely reasonable restriction — but rather to highlight how the NIST DRBGs do not fit easily into these security models when allowed to produce many output blocks per request. As such, the impact of state compromise *within a* generate *call* has been largely overlooked up until now. A notable exception is in a observation by Bernstein [16], (made concurrently to the production of this work) which criticizes the inefficiency of CTR-DRBG's update function which must be applied for forward-security. This is used as a motivating example for the use of 'fast-key erasure RNGs', which erase their key as soon as it is used, and thus this work does not create overlap without subsequent discussion.

Efficiency. The work by Bernstein [16] raises an important point — the extra block cipher calls incurred by the state update process of the CTR-DRBG certainly makes generating many blocks of output in a single request and buffering them for use in other processes an appealing implementation decision. Indeed the SP 800-90A standard says of the performance of the CTR-DRBG "For large generate requests, CTR-DRBG produces outputs at the same speed as the underlying block cipher algorithm encrypts data. Furthermore, CTR-DRBG is parallelizable.", reflecting how generating and buffering large numbers of output blocks per request maximizes the efficiency of output generation relative to the number of state updates. The case is similar for the HMAC-DRBG and HASH-DRBG.

Attack scenario. In the remainder of the section, we consider an attacker who manages to compromise part of the state within a generate call. As discussed in Section **??**, by this we mean that during some generate call with input tuple $(S, \beta, addin)$, the attacker compromises part of one of the intermediate states S^j for $j \in [1, m]$ where $S^0 \leftarrow \mathsf{init}(S, \beta, addin)$, and $(r^j, S^j) \leftarrow \mathsf{next}(S^{j-1})$ for $j = 1, \ldots, m$. We note that for each of the NIST DRBGs, if additional input is not used in the call then init returns the state unchanged, $S^0 = S$. This creates a greater window of opportunity in which this state may be compromised, as it will be set in memory following the conclusion of the previous generate call. As we shall see, not using additional input simplifies all attacks described, making state compromise in this case especially troubling.

We assume the DRBG in question is being used to produce multiple output blocks per request in order to fill a buffer. Some of the buffered output may be used for public values such as nonces, whereas other parts may be used for secret values such as keys or Diffie-Hellman exponents. In particular, if an attacker can use partial state information in conjunction with an output block sent in the clear as a nonce to recompute the unseen output, then they may be able to recover output blocks used as security critical secrets, thus breaking the security of the consuming application. After describing attacks against the NIST DRBGs, we will briefly discuss how work by Cohney et al. [22], which details how to exploit the compromised ANSI X9.31 PRG within a TLS implementation, can be extended to the case in which a compromised NIST

DRBG is used.

Security goals. We take the general view that for a good DRBG, all security critical state variables must be known to compute future unseen output[5], and recovery of past output blocks should never be feasible even if the entire secret state is known. As we shall see, none of the NIST DRBGs satisfies both of these properties when state information may be leaked within a generate call, with the CTR-DRBG faring especially badly.

A. Security of the CTR-DRBG with a Compromised Key

In this section, we describe how the invertibility of the block cipher used by the CTR-DRBG DRBG — and the fact that each output block in a given generate call is an encryption of the secret counter V — makes leakage of the secret key component of the internal state especially damaging.

While such an observation about block cipher based DRBGs is not new (see for example [40], [25], [22] for discussion of attacks against the block-cipher based ANSI X9.31 DRBG), we have not seen a treatment of state recovery attacks against the CTR-DRBG before.

Recovery of past / future output with a known key. The evolution of the state of the CTR-DRBG within a generate call is depicted in Figure 2 for an initial state $S = (K, V, cnt)$ and block cipher $\mathsf{E} : \{0,1\}^\kappa \times \{0,1\}^\ell \rightarrow \{0,1\}^\ell$. If additional input is used in the call, it will be incorporated into the state variables during the init subroutine via a call to update; otherwise they are left unchanged. Either way, we denote the state variables used during the iterative output generation process as K^0, V^0.

To satisfy a request for β bits of pseudorandom output, the CTR-DRBG generates $m = \lceil \beta/\ell \rceil$ output blocks by computing

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

for $j = 1, \ldots, m$, and addition is understood to be modulo 2^ℓ. The resulting output blocks are concatenated and returned as output $R_q = \text{left}(r^1 || \ldots || r^m)$, where we emphasize that the standard permits m to be as large as 2^{12}.

Notice that the key K^0 *never updates* through these m iterations. As such an attacker who is able to compromise the key component of any of the intermediate states will learn K^0, the key used in the production of every block of output in that call. To recover the internal counter, all that is then required is a single (and arbitrary) output block r^k produced during the same output generation request. The attacker can simply decrypt this output block to recover the underlying counter:

$$V^0 + k = \mathsf{D}(K^0, r^k) \ ,$$

from which recovering the initial state counter V^0 is trivial[6]. The attacker can then recompute *all* unseen output blocks produced within the generate call — including output blocks produced *prior* to the compromised state / output block — by computing

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

for $j = 1, \ldots, m$. If the output learnt by $\mathcal{A}$ is an incomplete block, $\mathcal{A}$ can iterate through possibilities for the remaining bits; recovering a second output block produced at any point within the generate call will allow $\mathcal{A}$ to verify his guesses with accuracy close to 1.

In our attack scenario, this instantly provides the attacker with *all* pseudorandom output used to fill the buffer — up to $2^{19} - \ell = 262,016$ bits of unseen output — having only learnt part of the state, including that produced *prior* to the point of state compromise.

We now consider the recovery of future output. If additional input is not used by the implementation, then the deterministic update function which constitutes the final subroutine at the conclusion of the generate calls depends only on K^0 and $V^0 + m$ — both of which are known to $\mathcal{A}$. In this case, $\mathcal{A}$ can immediately recover the updated state S' and then run the generator forwards to recover *all future output* up until the next high entropy reseed (which may not be mandated for another $2^{48} - 1$ generate calls). If additional input is used, then the attacker will need to guess both the additional input incorporated into the state at the conclusion of the compromised generate call, along with that used in each subsequent generate call in order to execute the same attack. However, since the standard allows additional input to be low entropy or even public, this is certainly achievable for such implementations.

Summary. By allowing the attacker to compute both past and future output blocks given leakage of only part of the secret state means that the CTR-DRBG falls down on both of our criteria. The fact that all output from a given generate call can be recovered given an arbitrary output block in the event of key leakage is especially damaging, since any output blocks used as e.g., secret keys generated in the same call can be recovered irrespective of their position relative to the block learnt by the attacker.

In comparison, the infamously backdoored DualEC-DRBG only allowed recovery of output produced *after* the block learnt by the attacker, impacting its practical exploitability in protocols [20], [21]. Indeed the latter work describes how in the (presumed backdoored) Juniper Screen OS IKE implementation, nonces and Diffie-Hellman exponents appear to be generated in a somewhat unnatural order to sidestep this issue. In contrast for the CTR-DRBG generating both a

[5]If state variables are partially known, the amount of work required to compute unseen future output should be roughly equivalent to brute-forcing the unknown bits.

[6]We assume throughout this section that the attacker has sufficient knowledge of the implementation to know the position within the sequence of output blocks at which the compromised block lies. If this is not the case, it is easy to see how encrypting all $V^j \in [V^k - m + 1, V^k + m - 1]$ guarantees all output blocks will be recovered successfully. The case of the other DRBGs is similar.

nonce and secret key in the same generate call, the order in which they were generated is irrelevant, greatly enhancing the exploitability of the compromised CTR-DRBG.

B. Security of the HMAC-DRBG with a Compromised Key

In this section we describe how compromise of the key component of the state of the HMAC-DRBG at any point during a generate call, in conjunction with a single output block from the same call, facilitates the recovery of all output produced after the compromised block. However on a more positive note, computation of any past output blocks — even given the entire secret state — appears to be infeasible.

We also describe how, in the case that additional input is not used, the (seemingly fairly superficial) distinguishability of the final output block produced in the last call prior to state compromise discussed in Section IV-B in fact offers a chance to extend the attack.

Recovery of future output with a known key. The evolution of the state of the HMAC-DRBG within a generate call is depicted in Figure 3 for an initial state $S = (K, V, cnt)$. If additional input is used in the call, it will be incorporated into the state variables during the init subroutine via a call to update; otherwise they are left unchanged. Either way, we denote the state variables used during the iterative output generation process as K^0, V^0. The output blocks required to satisfy the request are then generated by computing

$$r^j = \mathsf{HMAC}(K^0, V^{j-1})$$

for $j = 1, \ldots, m$ where in each iteration the internal counter V^j is updated to the most recent output block produced; as such $V^j = r^j$ for $j = 1, \ldots, m$.

Throughout this process, the key K^0 *never updates*. As such if the key component of any of the intermediate states is compromised, the attacker will learn the key K^0 used to produce all output in that generate call. With this in place, the attacker need only compromise a single output block r^k for some $k \in [1, m]$ produced in the same generate call — which by construction will be equal to the internal counter V^k — to recover all subsequent output blocks generated in that call as

$$r^j = \mathsf{HMAC}(K^0, r^{j-1})$$

for $j = k + 1, \ldots, m$. If additional input is not used, then — armed now with both inputs to the deterministic update function which constitutes the final subroutine at the conclusion of the call — the attacker can immediately recover the new state S' and run the DRBG forward to recover all subsequent output values. If additional input is used, then the attacker will need to guess both the additional input string incorporated into the state at the conclusion of the compromised generate call, along with that used in each subsequent generate call in order to compute output from subsequent generate calls. Guesses can be tested against output captured from the following call.

Connection to forward security. It was noted in Section IV-B that if additional input is not used, then the final output block produced in a call r^m is trivially distinguishable from random given the updated state at the conclusion of that call $S' = (K', V', cnt')$, since $V' = \mathsf{HMAC}(K', r^m)$.

While this may have appeared a rather pedantic criticism of the DRBG, and of minor practical concern, it is interesting to note that it offers an extension of the above attack. Consider an attacker who learns the final block of output r^m produced in a given generate call, and additionally manages to learn the key K' used for output generation in the following call. Then since the key K' does not update in this period (recall if additional input is not used, the init subroutine performed at the start of each generate call returns the state unchanged), the attacker can recover the remaining secret state variable as $V' = \mathsf{HMAC}(K', r^m)$, and correspondingly compute all subsequent output in both this and future calls. Since we have assumed that additional input is not used in this implementation, no additional effort is required to execute the attack. This highlights the power of security proofs to surface subtle security flaws in an algorithm.

Security of past in a compromised generate **call.** On a more positive note, it would appear that even if an attacker learns the *entire* state $S^k = (K^0, V^k, cnt)$ of the HMAC-DRBG at some point within the generate call, it is infeasible recover the unseen output blocks r^j for $j \in [1, k-1]$ which were produced in the call prior to the point of compromise. We describe the intuition for this in the case that additional input is not used (the case in which it is used is similar). Letting K, V denote the state variables at the commencement of the compromised generate call, where $V = \mathsf{HMAC}(K, V^*)$ and V^* denotes the final output block / counter from the previous call, then for $j = 1, \ldots, m$ we have that

$$r^j = V^j = \mathsf{HMAC}^{j+1}(K^0, V^*) \,, \tag{1}$$

where $\mathsf{HMAC}^i(K^0, \cdot)$ denotes the i^{th} iterate of $\mathsf{HMAC}(K^0, \cdot)$ on the given input. As such, to recover r^j given K^0, V^k where $j < k$, corresponds to finding preimages of $\mathsf{HMAC}(K^0, \cdot)$.

Since the key K^0 is known to the attacker, we clearly cannot argue that this is difficult based on the PRF-security of HMAC. However, that HMAC with a *known key* behaves like a random oracle is a fairly common assumption (see e.g., [42], [34], [11], [28]). This, coupled with the fact that the underlying input V^* in equation 1 is the result of an HMAC computation under a secret and random key in the previous call — and so by definition pseudorandom and unpredictable — makes it reasonable to assume that computing the target preimages is infeasible. Formalizing this intuition under a standard model assumption remains an interesting open question.

C. Security of the HASH-DRBG DRBG with a compromised counter.

We now describe how compromise of the counter component of the state of the HASH-DRBG at any point during a generate call facilitates the recovery of all output produced in that call, without any need to additionally compromise an output block. On the positive side, the use of the constant C

during state updates seems to contain the damage to a single generate call.

Recovery of all output within a generate call with a compromised counter. The evolution of the state of the HASH-DRBG within a generate call is depicted in Figure 4 for an initial state $S = (V, C, cnt)$ and hash function $\mathsf{SH} : \{0,1\}^* \rightarrow \{0,1\}^\ell$. Letting V^0 denote the counter variable possibly updated with additional input at the start of the generate call, then to satisfy a request for β of output, the generator computes

$$r^j = \mathsf{SH}(V^0 + j) \tag{2}$$

for $j = 1, \ldots m$ where $\lceil \beta/\ell \rceil$ and additional is understood to be modulo 2^{len}. It is easy to see that if the attacker can learn the internal counter $V^k = V^0 + k$ for $k \in [1, m]$ of the HASH-DRBG at any point within the generate call, then he can easily recover the initial counter V^0 recompute all unseen output from within that call as per equation 2.

Taking the HASH-DRBG instantiated with SHA-256 as an examples, this corresponds to up to 2^{24} bits of unseen output — including all output produced prior to the point of compromise. Unlike the attacks against the CTR-DRBG and HMAC-DRBG, the attacker need not compromise any output from the generate call in order to execute the attack; however if only part of the counter V^0 is learnt and $\mathcal{A}$ is forced to guess the unknown bits, then $\mathcal{A}$ would need to capture some of this output in order to test guesses.

Security of future output. Interestingly — unlike the other NIST DRBGs— recovering V alone is insufficient to run the DRBG forward and compute output from future generate calls. This is due to the fact that at the conclusion of the compromised generate call, the new state will be updated as

$$V' = (V + \mathsf{SH}(0\text{x}03||V) + C + cnt) \bmod 2^{len} ,$$

and, for all but the first generate call, it would appear to be infeasible to extract C from V without inverting the hash function. (The exception with the first generate call is because C is derived deterministically from the initial state variable V_0 during the setup process). That said, if an attacker can additionally compromise the counters V and V' from two consecutive generate calls in the case that additional input is not used, then he can easily recover C by calculating

$$C = (V' - \mathsf{SH}(0\text{x}03||V) - cnt) \bmod 2^{len} ,$$

thus facilitating the recovery of all subsequent output. If additional input is used, the same recovery of C is possible, albeit for a bit more more since th additional input string used in the second generate call must be guessed also.

Summary.

D. Comparison to ANSI X9.17/X9.31.

The ANSI X9.17 / X9.31 PRG [37] is a legacy block cipher-based PRG. Like the CTR-DRBG, the ANSI X9.17 / X9.31 PRG produces a single output block per iteration by encrypting the state of the PRG under a key K.[7] A time stamp is incorporated with the production of *each block*. It is well known that if the key is known, then it is possible recover the state of the PRG, with attacks being described in work such as [40], [25] and more recently by Cohney et al. [22]. As described in the latter work, in addition to the key, the attacker requires a complete output block (to decrypt an recover the internal state), plus sufficiently many bits of the following block in order to test time stamp guesses. Past / future output blocks can then be recovered, subject to the attacker's ability to guess the time stamp used in the production of each target block.

Attacking compromised DRBGs in TLS implementations. Checkoway et al. in describe how the Dual EC DRBG backdoor may be exploited in real world TLS [21] implementations, which Cohney et al. [22] extend to exploit a known key attack against the ANSI X9.31 PRG in a TLS handshake. We briefly recall their findings, and discuss how the NIST DRBGs fare in the same context.

In the case of the TLS 1.0, 1.1, or 1.2 handshake, a 32-byte random nonce is sent along with the client / server hello messages. After establishing a cipher suite, the client and server negotiate a shared secret accordingly by e.g., exchanging Diffie-Hellman shares. If the nonce contains a full block of ANSI X9.31 output (plus enough of the following block to guess time stamp guesses), this is sufficient to execute the state recovery described by Cohney et al. [22].

To see how this attack applies to the NIST SP 800-90 DRBGs, suppose that the 256-bit nonce contains a single CTR-DRBG output block (128-bits) or HMAC-DRBG output (256-bits), and that the secret keying material was generated within the same generate call and buffered. Then an attacker who learns the output production key K^0, can execute the attacks described in Sections V-A, V-B to recover the secret keying material. Worse still, for these algorithms there is *no need* to guess any additional input, as once incorporated at the beginning of a generate call, the (up to 2^{19}) bits of output generated within that call are produced entirely deterministically. Even if the output of the PRG is not buffered, the attacker can still run the DRBG forward and recover the keying material generated later in the handshake subject to guessing the (low-entropy) additional inputs. Each correct guess, facilitates the recovery of the multiple blocks of output produced in that call. As such, a CTR-DRBG / HMAC-DRBG implementation which is susceptible to side channel attacks functions similarly to an ANSI X9.31 with a poorly generated key, or the backdoored DualEC-DRBG.

VI. Security of Additional Input

In this section, we describe how implementing the CTR-DRBG without a derivation function can make it substantially easier to recover strings of additional input fed to

[7] The X9.17 / X9.31 PRG differs from the CTR-DRBG in that the key K does not update at all over successive generate calls, and as such is not forward secure. The CTR-DRBG is certainly a significant improvement in this important regard.

the DRBG, in the event that the key component of the state is compromised. This is particularly concerning, since the standard allows these additional input strings to contain secrets and sensitive data as long as they are not protected at a higher strength than the implementation (see Section III). As such, they may contain secret or private information such as social security numbers, PINs and passwords.

Use of a derivation function. As detailed in Section III-A, the standard allows the CTR-DRBG DRBG to be implemented with or without a derivation function. In the former case, additional input strings and entropy inputs are first conditioned with the block cipher derivation function CTR-DRBG df before being XORed into the state of the DRBG, whereas if the derivation function is now used, these raw inputs are XORed in directly.

One can verify from the pseudocode description of the CTR-DRBG df in Appendix C that to derive a $(\kappa + \ell)$-length string from a T-bit input, requires N block cipher computations where

$$N = \lceil (\kappa + \ell)/\ell \rceil \cdot \left(\lceil (T + 72)/\ell \rceil + 2 \right) ,$$

where κ and ℓ denote the key and block size of the block cipher respectively. Since this computation is required for every generate call which includes additional input on top of every reseed and setup, this represents a significant overhead — especially given that the standard permits additional input strings to be up to $T = 2^{35}$ bits in length. Indeed a set of slides on the NIST DRBGs by Kelsey from 2004 [39] includes the comment "Block cipher derivation function is expensive and complicated... When gate count or code size is an issue, nice to be able to avoid using it!" As such, it is easy to see the appeal of implementing the CTR-DRBG without a derivation function.

Recovery of additional input. We first describe the additional input recovery attack against the CTR-DRBG implemented without a derivation function in the ideal attack conditions, and then discuss how to extend this to more general cases.

Consider two successive output generation requests, such that the string of additional input used in the first call $addin_1$ is either known to the attacker, or is equal to ε (that is to say, additional input was not used in that call). Suppose further that the string of additional input $addin_2$ used in the second call is of the form $addin_2 = X_1 || X_2$ where $X_1 \in \{0,1\}^\kappa$ is known to the attacker, and $X_2 \in \{0,1\}^\ell$ consists of ℓ unknown bits, which include a secret value such as a password which will be the target of the attack. (Recall that κ and ℓ denote the key and block size of the underlying block cipher respectively.) Let $S_0 = (K_0, V_0, cnt_0)$ and $S_1 = (K_1, V_1, cnt_1)$ denote the state of the CTR-DRBG at the commencement of the first and second call respectively, where adding a superscript of 0 to those state variables indicated that they have been updated with additional input via the init subroutine at the start of that generate call.

Now suppose an attacker has executed the state recovery attack described in Section V-A against the first generate call, and so knows the state variables K_0^0, V_0^0. Since $addin_1$ is known to the attacker by assumption, he knows all inputs to the final subroutine, and so can immediately compute the updated state variables K_1, V_1 used in the following call.

At the beginning of the second generate call, the state components K_1, V_1 are first updated with the additional input string $addin_2$ via $(K_1^0, V_1^0) \leftarrow \mathsf{update}(addin_2, K_1, V_1)$ during the init subroutine. The update function first computes $C_i \leftarrow \mathsf{E}(K_1, V_1 + j)$ for $j = 1, \ldots, m$ where $m = \lceil \frac{\kappa+\ell}{\ell} \rceil$, sets $K_1^* || V_1^* \leftarrow \text{left}(C_1 || \ldots || C_m)$, before computing the updated key / counter K_1^0 / V_1^0 as

$$K_1^0 || V_1^0 \leftarrow K_1^* || V_1^* \oplus addin_2 .$$

Since $addin_1 = X_1 || X_2 \in \{0,1\}^{\kappa+\ell}$ where $X_1 \in \{0,1\}^\kappa$ is known to $\mathcal{A}$, it follows that the updated key

$$K_1^0 = K_1^* \oplus X_1$$

is known to $\mathcal{A}$ also. The (unknown) counter V_1^0 is of the form

$$V_1^0 = V_1^* \oplus X_2$$

where V_1^* is again known by $\mathcal{A}$, and $X_2 \in \{0,1\}^\ell$ contains the target secret.

Output during the second generate request is computed by encrypting the iterating counter under the fixed key K_1^0; as such each block of output r^k produced in the call will be of the form

$$r^k = \mathsf{E}(K_1^0, V_1^0 + k) = \mathsf{E}(K_1^0, (V_1^* \oplus X_2) + k) ,$$

for $k \in [1, \lceil \frac{\beta}{\ell} \rceil]$ and β denotes the number of bits of output requested in the call. Suppose that $\mathcal{A}$ learns a single complete output block produced in the request, used for example as a nonce. Then $\mathcal{A}$ can recover the target secret X_2 — consisting of 128-bits of unknown and secret data — with probability one by using the known key K_1^0 to decrypt the output block, and XOR-ing in the known counter value V_1^*:

$$X_2 = (\mathsf{D}(K_1^0, r^k) - k) \oplus V_1^* .$$

Extensions. We have discussed the ideal attack conditions here; however the same attack is still possible even if $addin_1$ and / or $X_1 = \text{left}(addin_2, \kappa)$ are not known to the attacker. Supposing these components have γ_1, γ_2 bits of entropy respectively, then repeating the above process for each possible pair of candidate values will recover the correct secret $X_2 = \text{select}(addin_2, \kappa + 1, \ell)$ among a list of $2^{\gamma_1+\gamma_2}$ candidates. If the data in X_2 is of a distinctive structure, or multiple output blocks are learnt from the generate call, this can help $\mathcal{A}$ quickly eliminate candidates. Either way, the entropy of X_2 is reduced to $\gamma_1 + \gamma_2$-bits, a loss which — given X_2 contains up to 128 bits of unknown data — may be substantial.

Security benefit of the derivation function. The attack exploits the way in which raw additional input is XORed directly into the state of the DRBG; as such the structure of the additional input is preserved, and any entropy within it is not mixed into both key and counter. In contrast when the CTR-DRBG derivation function is used to condition the

input before it is XORed into the state, the structure of the data is sufficiently destroyed that the above attack no longer works. Indeed, even if the attacker could compromise the raw derivation function output, it is difficult to see how they could recover the underlying additional input string without exhausting the entropy of that string in a brute-force attack. The cases of the HASH-DRBG and HMAC-DRBG, where additional input is either hashed prior to its incorporation into the state or used as an input to HMAC are similar.

VII. Conclusion

We conducted a multi-layered and formal analysis of the three DRBG algorithms in the NIST SP 800-90A standard. On the positive side, we verified the forward security of each of the DRBG mechanisms as claimed in the standard. However we argue that the usual security models do not adequately capture the somewhat unconventional design of these DRBGs. Taking a closer look, we uncovered a number of problems with the design of these algorithms.

The key cause for concern is that in the event of an attacker recovering a certain part of the internal state of the DRBG, the security of the algorithms breaks down in unexpected ways. The result of this is that an attacker can recover unseen output — and correspondingly break the security of any consuming application relying on this output for its secrets. While each of the algorithms admits varying degrees of security failure, the CTR-DRBG fares especially badly in the event that the key component of the state is compromised, due to the invertibility of the block cipher.

Furthermore, if the design choice to implement the CTR-DRBG without a derivation function is taken, this allows an attacker to extend the above attack to recover strings of additional input fed to the DRBG from public output. This is especially worrying since the standard allows these strings to contain secrets.

Flexibility in algorithm specifications. The flaws that we have pointed out in this paper can be viewed as algorithm specifications that are overly flexible, allowing both implementers and users of these algorithms to make design choices that demonstrably weaken the claimed security properties. In particular, it is the flexibility to request variable amounts of random bits per generate call which allows a partial state compromise at any point during these calls to weaken all subsequent output produced in the call — and in the case of the CTR-DRBG and HMAC-DRBG, all output up to the next high entropy reseed. The option to not include additional input in output generation requests can only make these attacks easier. Furthermore, the flexibility in the specification that allows an implementation of the CTR-DRBG to omit the derivation function, opens up the possibility that additional input may be recovered. As such, these vulnerabilities may be a warning to standard writers to avoid unnecessary flexibility as it may lead to unintended security vulnerabilities.

Recommendations. Fortunately, because these vulnerabilities arise from choices that are allowed by the specification, we may offer recommendations to make the use of these algorithms more secure. First off, if the algorithms are being run in a setting where side channel attacks — potentially leading to partial state recovery — are of particular concern, we can recommend the HASH-DRBG as the safest choice of a generator. Furthermore, the CTR-DRBG derivation function should always be used. Additional input should be (safely) incorporated in output generation requests wherever possible, and of course the DRBG should be 'properly' reseeded with fresh entropy as often as is practical.

While the standard allows outputs of variable and sizeable length to be requested in each generate call, users of these algorithms should not 'batch up' calls by making a single call for all randomness required and separating the randomness into separate values. For example, although it is faster to generate all the randomness required in a TLS handshake up front, this introduces security vulnerabilities.

More generally, using the technique of "fast key erasure" [16] to fill a buffer with randomness, taking it out subsequently when needed, introduces security vulnerabilities. The most secure way to call this algorithm would be to use it to generate one block of output at a time, and updating the state with fresh entropy as often as is feasible, although this may introduce significant performance degradation over the algorithms as standardized. While each of these decisions may introduce additional cost, there are clear security benefits to avoiding the unnecessary flexibility allowed by the standard. We conclude with some open problems, and directions for further work.

Analysis of setup **and** reseed**.** We have analyzed the NIST DRBGs operating as deterministic PRGs. Extending the analysis to model high entropy reseeds within the robust pseudorandom number generator with input (PRNG) framework of Dodis et al [27], and extended to better capture 'real world' PRNGs in [51], is an important direction for future work. Similarly, analyzing how well a state output by the setup algorithm matches the 'idealized' states we have assumed here, is an important extension of our results.

Optimizing security and efficiency. The design flexibilities we critique above are generally related to efficiency savings. Designing DRBGs which achieve an optimal balance between security and efficiency, represents a key direction for future work. For example, redesigning the CTR-DRBG derivation function so that it adequately extracts entropy from its inputs without such a computational overhead would make its use much more palatable in terms of efficiency.

The gap between the specification of these DRBGs, which allows for various optional inputs and implementation / usage choices, and the far simpler manner in which DRBGs are typically modeled in the literature, could indicate that theoretical models are not best capturing real world DRBGs as is. Extending these models to factor in these concerns, may help understand the limits and possibilities of what can be achieved here.

REFERENCES

[1] Michel Abdalla, Sonia Belaïd, David Pointcheval, Sylvain Ruhault, and Damien Vergnaud. Robust pseudo-random number generators with input secure against side-channel attacks. In *International Conference on Applied Cryptography and Network Security*, pages 635–654. Springer, 2015.

[2] Michel Abdalla and Mihir Bellare. Increasing the lifetime of a key: a comparative analysis of the security of re-keying techniques. In *Asiacrypt*, volume 1976, pages 546–559. Springer, 2000.

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

[4] E Barker and J Kelsey. Recommendation for random number generation using deterministic random bit generators, nist sp800-90a rev. 1. *Retrieved September*, 3:2016, 2015.

[5] Elaine Barker and John Kelsey. Nist draft special publication 800-90b recommendation for the entropy sources used for random bit generation. 2012.

[6] Elaine Barker, John Kelsey, Rebecca Blank, Acting Secretary, and Patrick D. Gallagher. Draft nist special publication 800-90c recommendation for random bit generator (rbg) constructions, 2012.

[7] Elaine B Barker and John M Kelsey. Sp 800-90a. recommendation for random number generation using deterministic random bit generators. 2012.

[8] William C Barker, Elaine Barker, et al. Recommendation for the triple data encryption algorithm (tdea) block cipher: Nist special publication 800-67, revision 2. 2012.

[9] Mihir Bellare. New proofs for nmac and hmac: Security without collision-resistance. In *Annual International Cryptology Conference*, pages 602–619. Springer, 2006.

[10] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Crypto*, volume 96, pages 1–15. Springer, 1996.

[11] Mihir Bellare, Thomas Ristenpart, and Stefano Tessaro. Multi-instance security and its application to password-based cryptography. In *Advances in Cryptology–CRYPTO 2012*, pages 312–329. Springer, 2012.

[12] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[13] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. In *Advances in Cryptology–EUROCRYPT*, volume 4004, page 10, 2006.

[14] Mihir Bellare and Bennet Yee. Forward-security in private-key cryptography. In *CT-RSA*, volume 2612, pages 1–18. Springer, 2003.

[15] Daniel J Bernstein. Cache-timing attacks on aes, 2005.

[16] Daniel J. Bernstein. Fast-key-erasure random-number-generators, 2017.

[17] Daniel J Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko Van Someren. Factoring rsa keys from certified smart cards: Coppersmith in the wild. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 341–360. Springer, 2013.

[18] Andrey Bogdanov. Improved side-channel collision attacks on aes. In *Selected Areas in Cryptography*, volume 4876, pages 84–95. Springer, 2007.

[19] Matthew J Campagna. Security bounds for the nist codebook-based deterministic random bit generator. *IACR Cryptology ePrint Archive*, 2006:379, 2006.

[20] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the juniper dual ec incident. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 468–479. ACM, 2016.

[21] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J Bernstein, Jake Maskiewicz, Hovav Shacham, Matthew Fredrikson, et al. On the practical exploitability of dual ec in tls implementations. In *USENIX security symposium*, pages 319–335, 2014.

[22] Shaanan Cohney, Matthew D. Green, and Nadia Heninger. Practical state recovery attacks against legacy rng implementations.

[23] Mario Cornejo and Sylvain Ruhault. Characterization of real-life prngs under partial state corruption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1004–1015. ACM, 2014.

[24] Jean Paul Degabriele, Kenneth G Paterson, Jacob CN Schuldt, and Joanne Woodage. Backdoors in pseudorandom number generators: Possibility and impossibility results. In *Annual Cryptology Conference*, pages 403–432. Springer, 2016.

[25] Anand Desai, Alejandro Hevia, and Yiqun Yin. A practice-oriented treatment of pseudorandom number generators. In *Advances in Cryptology—EUROCRYPT 2002*, pages 368–383. Springer, 2002.

[26] Yevgeniy Dodis, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart. A formal treatment of backdoored pseudorandom generators. In *EUROCRYPT (1)*, pages 101–126, 2015.

[27] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input:/dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 647–658. ACM, 2013.

[28] Yevgeniy Dodis, Thomas Ristenpart, John P Steinberger, and Stefano Tessaro. To hash or not to hash again?(in) differentiability results for h 2 and hmac. In *CRYPTO*, volume 7417, pages 348–366. Springer, 2012.

[29] Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. How to eat your entropy and have it too – optimal recovery strategies for compromised rngs. In *CRYPTO*, volume 2014. Springer, 2014.

[30] Peter Gaži, Krzysztof Pietrzak, and Michal Rybár. The exact prf-security of nmac and hmac. In *International Cryptology Conference*, pages 113–130. Springer, 2014.

[31] Ian Goldberg and David Wagner. Randomness and the netscape browser. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 21(1):66–71, 1996.

[32] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, volume 8, 2012.

[33] Shoichi Hirose. Security analysis of drbg using hmac in nist sp 800-90. In *WISA*, pages 278–291. Springer, 2008.

[34] Burt Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. 2000.

[35] Wilson Kan. Analysis of underlying assumptions in nist drbgs. *IACR Cryptology ePrint Archive*, 2007:345, 2007.

[36] Q Ye Katherine, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W Appel. Verified correctness and security of mbedtls hmac-drbg. 2017.

[37] Sharon S Keller. Nist-recommended random number generator based on ansi x9. 31 appendix a. 2.4 using the 3-key triple des and aes algorithms. *NIST Information Technology Laboratory-Computer Security Division, National Institute of Standards and Technology*, 2005.

[38] J Kelsey. Five drbg algorithms based on hash functions and block ciphers, July 2004.

[39] John Kelsey. Five drbg algorithms based on hash functions and block ciphers. *National Institute of Standards and Technology*, 2004.

[40] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption*, pages 168–188. Springer, 1998.

[41] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[42] Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. In *CRYPTO*, volume 6223, pages 631–648. Springer, 2010.

[43] Arjen Lenstra, James P Hughes, Maxime Augier, Joppe Willem Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, whit is right. Technical report, IACR, 2012.

[44] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *International Conference on Information Security and Cryptology*, pages 343–358. Springer, 2002.

[45] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[46] Colin Percival. Cache missing for fun and profit, 2005.

[47] Nicole Perlroth. Government announces steps to restore confidence on encryption standards. 2013.

[48] NIST FIPS Pub. 197: Advanced encryption standard (aes). *Federal information processing standards publication*, 197(441):0311, 2001.

[49] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.

[50] Sylvain Ruhault. Sok: Security models for pseudo-random number generators. *IACR Transactions on Symmetric Cryptology*, 2017(1):506–544, 2017.

[51] Thomas Shrimpton and R Seth Terashima. A provable security analysis of intel's secure key rng. In *EUROCRYPT (1)*, pages 77–100, 2015.

[52] Thomas Shrimpton and R Seth Terashima. Salvaging weak security bounds for blockcipher-based constructions. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*, pages 429–454. Springer, 2016.

[53] Dan Shumow and Niels Ferguson. On the possibility of a back door in the nist sp800-90 dual ec prng. In *Proc. Crypto*, volume 7, 2007.

[54] James M Turner. The keyed-hash message authentication code (hmac). *Federal Information Processing Standards Publication*, 2008.

[55] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 debian openssl vulnerability. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 15–27. ACM, 2009.

APPENDIX

A. Examples of Parameter Settings

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

Fig. 6: Table showing parameter settings for the NIST DRBGs described in Section III. All quantities are given in bits.

1) Theorem 4.1: **Proof:** We present the proof in the case that additional input is used; the case in which additional input is not used can easily be recovered from this proof by omitting the extra state update at the start of each generate call which is only executed when additional input is present.

We shall repeatedly reference the pseudocode in Figure 7, which depicts the production of output / state variables in Game $\mathrm{Fwd}^{\mathcal{A},q}_{\mathsf{CTR\text{-}DRBG},\beta}$ with challenge bit $b = 0$, and where each of the q iterations of the for loop corresponds to a generate call.

We begin by defining two sequences of hybrid games G^j for $j \in [0, q]$ and $j \in [\overline{0}, \overline{q}]$. These are defined such that game G^0 is equivalent to Game $\mathrm{Fwd}^{\mathcal{A},q}_{\mathsf{CTR\text{-}DRBG},\beta}$ with challenge bit $b = 0$ (e.g., the adversary receives all real outputs in his challenge), and for $j \in [1, q]$ game G^j is the same as game G^{j-1} except during the j^{th} generate call / iteration we replace the output R_j (line 20) and the state variables as updated at the conclusion of the call V_j, K_j, (line 27) with uniform bit strings of appropriate length. Similarly for $j \in [\overline{0}, \overline{q}]$, we let game $G^{\overline{0}}$ be identical to game Game $\mathrm{Fwd}^{\mathcal{A},q}_{\mathsf{CTR\text{-}DRBG},\beta}$ with challenge bit $b = 1$ (so the adversary receives all random outputs in his challenge), and then define $G^{\overline{j}}$ to be the same as game $G^{\overline{j-1}}$

proc. main

1: $\mathcal{Q} \leftarrow (addin_1, \ldots, addin_q)$
2: $K_0 || V_0 \leftarrow\$ \{0,1\}^{\kappa+\ell}$
3: $cnt_0 \leftarrow 1$
4: **for** i = 1, ..., q **do**
5: **if** $addin_i \neq \varepsilon$ **then**
6: $addin_i \leftarrow \mathsf{df}(addin_i, \kappa + \ell)$
7: $temp_1 \leftarrow \varepsilon\ ; m_1 \leftarrow \lceil (\kappa + \ell)/\ell \rceil$
8: **for** $k = 1, \ldots, m_1$ **do**
9: $V_{i-1} \leftarrow (V_{i-1} + 1) \bmod 2^\ell$
10: $C_k \leftarrow \mathsf{E}(K_{i-1}, V_{i-1})$
11: $temp_1 \leftarrow temp_1 || C_k$
12: $temp_1 \leftarrow \mathrm{left}(temp_1, (\kappa + \ell))$
13: $K^0_{i-1} || V^0_{i-1} \leftarrow temp_1 \oplus addin_i$
14: **else** $addin_i \leftarrow 0^{\kappa+\ell}\ ; (K^0_{i-1}, V^0_{i-1}) \leftarrow (K_{i-1}, V_{i-1})$
15: $temp_2 \leftarrow \varepsilon\ ; m_2 \leftarrow \lceil \beta/\ell \rceil$
16: **for** $k = 1, \ldots, m_2$ do
17: $V^k_{i-1} \leftarrow (V^{k-1}_{i-1} + 1)$ **mod** 2^ℓ
18: $r^k_i \leftarrow \mathsf{E}(K^0_{i-1}, V^{k-1}_i)$
19: $temp_2 \leftarrow temp_2 || r^k_i$
20: $R_i \leftarrow \mathrm{left}(temp_2, \beta)$
21: $temp_3 \leftarrow \varepsilon\ ; m_1 \leftarrow \lceil (\kappa + \ell)/\ell \rceil$
22: **for** $k = 1, \ldots, m_1$ **do**
23: $V^{m_2}_{i-1} \leftarrow (V^{m_2}_{i-1} + 1) \bmod 2^\ell$
24: $C'_k \leftarrow \mathsf{E}(K^0_{i-1}, V^{m_2}_{i-1})$
25: $temp \leftarrow temp_3 || C'_k$
26: $temp \leftarrow \mathrm{left}(temp_3, (\kappa + \ell))$
27: $K_i || V_i \leftarrow temp \oplus addin_i$
28: $cnt_i \leftarrow cnt_{i-1} + 1$
29: $b' \leftarrow\$ \mathcal{A}(R_1, \ldots, R_q, (K_q, V_q, cnt_q), \mathcal{Q})$
30: **return** $(b = b')$

Fig. 7: Pseudocode for proof of Theorem 4.1.

except that in the j^{th} generate call / iteration state variables V_j, K_j (line 27) with random bit string. It follows that

$$\begin{aligned}\mathsf{Adv}^{\text{fwd}}_{\mathsf{CTR\text{-}DRBG},\beta}(\mathcal{A}, q) &= |\Pr\left[G^0 \Rightarrow 1\right] - \Pr\left[G^{\overline{0}} \Rightarrow 1\right]| \\ &\leq \sum_{i=0}^{q-1} \Big(|\Pr\left[G^i \Rightarrow 1\right] - \Pr\left[G^{i+1} \Rightarrow 1\right]| \\ &\qquad + |\Pr\left[G^{\overline{i+1}} \Rightarrow 1\right] - \Pr\left[G^{\overline{i}} \Rightarrow 1\right]| \Big) .\end{aligned}$$

In order bound the gap between these games, we define for each $j \in [0, q-1]$ a further set of hybrids G^j_k for $k \in [0, 8]$, in which we successively replace outputs of the block cipher $\mathsf{E} : \{0,1\}^\kappa \times \{0,1\}^\ell \to \{0,1\}^\ell$ under different keys with random bits strings during $(j+1)^{\text{st}}$ generate call / iteration. We begin by bounding the gap between games G^j and G^{j+1} for $j \in [0, q-1]$. Fix $j \in [0, q-1]$, and let G^j_0 be equivalent to game G^j.

Next we define game G^j_0 to be identical to game G^j_0 except the block cipher outputs $C_k \leftarrow \mathsf{E}(K_j, V_j + k)$ in line 10 are replaced with uniform bit string $C_k \leftarrow\$ \{0,1\}^\ell$ for $k = 1, \ldots, m_1$. We shall repeatedly invoke a standard argument to bound variants of this transition; we give a detailed treatment in this initial case.

We claim that there exists an adversary $\mathcal{B}$ in the PRP distinguishing game against E running in time $T \approx T'$ such that

$$|\Pr\left[G^j_0 \Rightarrow 1\right] - \Pr\left[G^j_1 \Rightarrow 1\right]|$$

$$\leq \mathsf{Adv}^{\mathrm{prp}}_{\mathsf{E}}(\mathcal{B}, m_1) + \frac{m_1^2}{2^{\ell+1}} \ .$$

To see this, notice that $\mathcal{B}$ can perfectly simulate the first j generate calls by choosing random output / state variable pairs $R_i||V_i||K_i \leftarrow\!\!\$\ \{0,1\}^{\beta+\ell+\kappa}$ for $i = 1, \ldots, j$. To simulate the $(j+1)^{\text{st}}$ generate call, $\mathcal{B}$ queries $V_j + 1, \ldots, V_j + m_1$ to his RoR oracle, receiving $C_1, \ldots, C_{m_1}$ in response. He sets $temp = \text{left}(C_1||\ldots||C_{m_1}, \kappa + \ell)$ in line 12, and uses these values to simulate the remainder of the game for $\mathcal{A}$. At the conclusion of the game he outputs 1 if and only if $\mathcal{A}$ does.

Notice that if $\mathcal{B}$'s oracle implements the real function, then he perfectly simulates game G_j^0; otherwise he perfectly simulates an intermediate game in which the block cipher E is replaced by a random permutation $\pi \leftarrow\!\!\$\ \text{Perm}(\ell, \ell)$. Moreover, $\mathcal{B}$ runs in time $T \approx T'$, and so the gap between these games may be bounded by a reduction to the PRP-security of E.

Now consider the intermediate game in which the block cipher E is replaced with the random permutation π, and notice that, since we iterate a counter with each query, the inputs to the random permutation π are distinct. As such the PRP/PRF-switching lemma (see e.g., [13] for a proof of this result) implies that we can define a further game in which we replace the permutation outputs $C_i \leftarrow \pi(V_j + k)$ with uniform bit strings $C_i \leftarrow\!\!\$\ \{0,1\}^{\ell}$, and that these games run identically *unless* two of the randomly sampled C_i collide (to see this, notice that since π is a permutation, the outputs produced by submitting the distinct queries to π will *never* collide). Since precisely m_1 such strings are sampled, it follows that the gap between these games is bounded above by $\frac{m_1^2}{2^{\ell+1}}$. Now this game is identical to game G_j^1, and so putting this altogether, it follows that

$$|\Pr\left[G_0^j \Rightarrow 1\right] - \Pr\left[G_1^j \Rightarrow 1\right]| \leq \mathsf{Adv}^{\mathrm{prp}}_{\mathsf{E}}(\mathcal{B}, m_1) + \frac{m_1^2}{2^{\ell+1}} \ ,$$

proving the claim. Notice that in this game, since $temp_1 \leftarrow \text{left}(C_1||\ldots||C_{m_1})$ (line 12) and $C_i \leftarrow \{0,1\}^{\ell}$ for $i = 1, \ldots, m_1$, this is equivalent to choosing $temp_1 \leftarrow\!\!\$\ \{0,1\}^{\kappa+\ell}$.

Next we define game G_2^j, in which instead of setting $K_j^0||V_j^0 \leftarrow temp_1 \oplus addin_{j+1}$ in line 13, we simply sample $K_j^0||V_j^0 \leftarrow\!\!\$\ \{0,1\}^{\kappa+\ell}$. Since in this game $temp_1 \leftarrow\!\!\$\ \{0,1\}^{\kappa+\ell}$, it follows that these games are identically distributed and so

$$\Pr\left[G_1^j \Rightarrow 1\right] = \Pr\left[G_2^j \Rightarrow 1\right] .$$

Next we define game G_3^j, which is identical to G_2^j except we replace the block cipher outputs used for output blocks (line 18), and the final state update (line 24) with random bits strings. By an analogous argument to that used previously, and noting that the block cipher key K^0 used for these encryptions is chosen uniformly in this game $K^0 \leftarrow\!\!\$\ \{0,1\}^{\kappa}$, a reduction to the PRP-security of E implies that there exists an adversary $\mathcal{C}$ in the PRP-distinguishing hame against E such that

$$|\Pr\left[G_3^j \Rightarrow 1\right] - \Pr\left[G_2^j \Rightarrow 1\right]| \leq \mathsf{Adv}^{\mathrm{prp}}_{\mathsf{E}}(\mathcal{C}, m_1 + m_2) + \frac{(m_1+m_2)^2}{2^{\ell+1}} \ .$$

The first term in the above equation follows since both G_2^j and an intermediate game in which the block cipher E is replaced by a random permutation π can be perfectly simulated by an attacker in the PRP-security game against E making $(m_1 + m_2)$ RoR oracle queries. The second term then follows since the $(m_1 + m_2)$ inputs to the random permutation are distinct, and so the PRP/PRF-switching lemma allows us to replace the permutation outputs with random bits strings incurring a loss of $\frac{(m_1+m_2)^2}{2^{\ell+1}}$. Notice that in this game, the output $R_{j+1} = \text{left}(C_1||\ldots||C_{m_2})$ in line 20, and $temp_3 = \text{left}(C_{m_2+1}||\ldots||C_{m_1+m_2}$ in line 26 — where $C_k \leftarrow\!\!\$\ \{0,1\}^{\ell}$ for $k = 1, \ldots, m_1 + m_2$ are the random bit strings introduced in this game hop — are now themselves equivalent to independent random bit strings.

Finally in game G_4^j, the state variables $K_{j+1}||V_j \leftarrow temp_3 \oplus addin_{j+1}$ in line 27 are replaced with random bit strings $K_{j+1}||V_{j+1} \leftarrow\!\!\$\ \{0,1\}^{\kappa+\ell}$. Again since $temp_3 \leftarrow\!\!\$\ \{0,1\}^{\kappa+\ell}$ in this game, it follows that the two games are identically distributed and so

$$\Pr\left[G_4^j \Rightarrow 1\right] = \Pr\left[G_3^j \Rightarrow 1\right] .$$

Furthermore, notice that since both output R_{j+1} and updated state variables K_{j+1}, V_{j+1} have been replaced with random bit strings in this game, we have that G_4^j is equivalent to game G^{j+1}. Putting this all together, we have that for each $j \in [0, q-1]$

$$|\Pr\left[G^j \Rightarrow 1\right] - \Pr\left[G^{j+1} \Rightarrow 1\right]| \leq \mathsf{Adv}^{\mathrm{prp}}_{\mathsf{E}}(\mathcal{B}, m_1) + \mathsf{Adv}^{\mathrm{prp}}_{\mathsf{E}}(\mathcal{C}, m_1 + m_2) + \frac{(m_1+m_2)^2 + m_1^2}{2^{\ell+1}} \ ,$$

completing the first part of the proof. With this in place, we now bound the gaps between games $G^{\overline{j+1}}$ and $G^{\overline{j}}$ for $j \in [0, q-1]$. We argue by a series of analogous steps to those above in reverse order, working towards returning the state variables K_j, V_j in each iteration to being computed honestly, while preserving the random output R_j. Since the steps are similar, we describe the first couple of steps more fully, and sketch the rest. We begin by fixing $j \in [0, q-1]$.

We begin by defining game G_5^{j+1} to be identical to game $G^{\overline{j+1}}$ (so the first $(j+1)$ outputs and updated state variables K_{j+1}, V_{j+1} are chosen at random, and the rest computed honestly). We will gradually alter the way in which the state is computed in the $(j+1)^{\text{st}}$ generate call. Next we define game G_6^{j+1} to be identical to G_5^{j+1} except rather than choosing the updated state variables K_{j+1}, V_{j+1} at random, we instead compute them as $K_{j+1}||V_{j+1} \leftarrow temp_3 \oplus addin_{j+1}$ where $temp_3 \leftarrow\!\!\$\ \{0,1\}^{\kappa+\ell}$. Clearly these games are identically distributed, and so

$$\Pr\left[G_6^{j+1} \Rightarrow 1\right] = \Pr\left[G_5^{j+1} \Rightarrow 1\right] .$$

Next we define game G_7^{j+1} to be identical to G_6^{j+1} except rather than sampling $temp_3 \leftarrow\!\!\$\ \{0,1\}^{\kappa+\ell}$, we instead

generate it by choosing by $K_j^0||V_j^0 \leftarrow\!\!\$\ \{0,1\}^{\kappa+\ell}$, computing $C_k = \mathsf{E}(K_j^0, V_j^0 + m_2 + k)$ for $k = 1, \ldots, m_1$, and setting $temp_3 = \text{left}(C_1||\ldots||C_{m_1}, \kappa + \ell)$ (see line 24). However, we still sample $R_{j+1} \leftarrow\!\!\$\ \{0,1\}^{\beta}$ (line 24), as opposed to computing this via the block cipher also. Looking ahead, this is allowed, since the output is chosen randomly and independently of the state variables in both games.

We may then define an adversary $\mathcal{B}'$ in the PRP-distinguishing game against E who proceeds as follows. $\mathcal{B}'$ perfectly simulates the first j calls by choosing $R_i||K_i||V_i \leftarrow\!\!\$\ \{0,1\}^{\beta+\kappa+\ell}$ for $i = 1, \ldots, j$. For the $(j+1)^{\text{st}}$ call, $\mathcal{A}$ chooses $R_{j+1} \leftarrow\!\!\$\ \{0,1\}^{\beta}$, and $V_j^0 \leftarrow\!\!\$\ \{0,1\}^{\ell}$, and then submits queries $V_j^0 + m_2 + k$ to his RoR oracle for $k = 1, \ldots, m_1$, receiving $C_1, \ldots, C_{m_1}$ in response. He sets $temp = \text{left}(C_1||\ldots||C_{m_1}, \kappa + \ell)$, and continues simulating the rest of the game as per the pseudocode. By an analogous argument to that used above using a reduction to the PRP-security of E and the PRP/PRF-switching lemma, and noting that the random permutation π is queried on precisely m_1 points, it follows that

$$|\Pr\left[G_7^{j+1} \Rightarrow 1\right] - \Pr\left[G_6^{j+1} \Rightarrow 1\right]| \leq \mathsf{Adv}^{\text{prp}}_{\mathsf{E}}(\mathcal{B}', m_1) + \frac{m_1^2}{2^{\ell+1}} .$$

In game G_8^{j+1}, we return to computing $K_j^0||V_j^0 \leftarrow temp_1 \oplus addin_{j+1}$ (see line 13) for $temp_1 \leftarrow\!\!\$\ \{0,1\}^{\kappa+\ell}$ rather than simply sampling these strings at random; clearly the two games are identically distributed. Finally in game G_9^{j+1} we generate $temp_1$ in line 12 via m_1 computations with the block cipher in CTR-mode with key / initial counter $K_j V_j \leftarrow\!\!\$\ \{0,1\}^{\ell}$ as sampled uniformly at the conclusion of the j^{th} generate call. Again by an analogous argument to that used previously, it follows that there exists an adversary $\mathcal{B}'$ in the PRP-security game against E such that

$$|\Pr\left[G_8^{j+1} \Rightarrow 1\right] - \Pr\left[G_7^{j+1} \Rightarrow 1\right]| \leq \mathsf{Adv}^{\text{prp}}_{\mathsf{E}}(\mathcal{B}', m_1) + \frac{m_1^2}{2^{\ell+1}} .$$

Now since we have returned to computing the state honestly in the $(j+1)^{\text{st}}$ call, while preserving the random output R_{j+1} game G_8^j is equivalent to game $G^{\overline{j}}$; therefore it follows that for each $j \in [\overline{0}, \overline{q-1}]$, there exists an adversary $\mathcal{B}$ running in time $T \approx T'$ such that

$$|\Pr\left[G^{\overline{j+1}} \Rightarrow 1\right] - \Pr\left[G^{\overline{j}} \Rightarrow 1\right]| \leq 2 \cdot \mathsf{Adv}^{\text{prp}}_{\mathsf{E}}(\mathcal{B}', m_1) + \frac{2m_1^2}{2^{\ell+1}} .$$

Putting this all together, we conclude that

$$\mathsf{Adv}^{\text{fwd}}_{\mathsf{CTR\text{-}DRBG},\beta}(\mathcal{A}, q) = 3q \cdot \mathsf{Adv}^{\text{prp}}_{\mathsf{E}}(\mathcal{B}, m_1) + q \cdot \mathsf{Adv}^{\text{prp}}_{\mathsf{E}}(\mathcal{C}, m_1 + m_2) + \frac{q \cdot (3m_1^2 + (m_1 + m_2)^2)}{2^{\ell+1}} .$$

proc. main

1: $\mathcal{Q} \leftarrow (addin_1, \ldots, addin_q)$
2: $K_0^*||V_0^* \leftarrow\!\!\$\ \{0,1\}^{\kappa+\ell}$
3: $cnt \leftarrow 1$
4: For $i = 1, \ldots, q$
5: If $addin_i \neq \varepsilon$
6: $K_{i-1}^0 \leftarrow \mathsf{HMAC}(K_{i-1}^*, V_{i-1}^*||\text{0x00}||addin_i)$
7: $V_{i-1}^0 \leftarrow \mathsf{HMAC}(K_{i-1}^0, V_{i-1}^*)$
8: $K_{i-1}^0 \leftarrow \mathsf{HMAC}(K_{i-1}^0, V_{i-1}^0||\text{0x01}||addin_i)$
9: $V_{i-1}^0 \leftarrow \mathsf{HMAC}(K_{i-1}^0, V_{i-1}^0)$
10: $temp \leftarrow \varepsilon$
11: For $k = 1, \ldots, m$
12: $V_{i-1}^k \leftarrow \mathsf{HMAC}(K_{i-1}^0, V_{i-1}^{(k-1)})$
13: $temp \leftarrow temp||V_{i-1}^k$
14: $K_i \leftarrow \mathsf{HMAC}(K_{i-1}^0, V_{i-1}^m||\text{0x00}||addin_i)$
15: $V_i \leftarrow \mathsf{HMAC}(K_i, V_{i-1}^m)$
16: If $addin_i \neq \varepsilon$
17: $K_i^* \leftarrow \mathsf{HMAC}(K_i, V_i||\text{0x01}||addin_i)$
18: $V_i^* \leftarrow \mathsf{HMAC}(K_i^*, V_i)$
19: $R_i \leftarrow \text{left}(temp, \beta)$
20: $cnt \leftarrow cnt + 1$
21: $b' \leftarrow\!\!\$\ \mathcal{A}(R_1, \ldots, R_q, K_q^*, V_q^*, \mathcal{Q})$
22: Return $(b = b')$

Fig. 8: Pseudocode for proof of Theorem 4.2.

B. Proofs from Section IV

1) Theorem 4.2: **Proof:** We present the proof in the case that additional input is used; the case in which additional input is not used can easily be recovered from this proof by omitting the extra HMAC computations which are only used when additional input is present.

We shall repeatedly reference the pseudocode in Figure 8, which depicts the production of output / state variables in Game $\text{Fwd}^{\mathcal{A},q}_{\mathsf{HMAC\text{-}DRBG}}$ with challenge bit $b = 0$, and where each of the q iterations of the For loop corresponds to a generate call.

We begin by defining two sequences of hybrid games G^j for $j \in [0, q]$ and $j \in [\overline{0}, \overline{q}]$. These are defined such that game G^0 is equivalent to Game $\text{Fwd}^{\mathcal{A},q}_{\mathsf{HMAC\text{-}DRBG}}$ with challenge bit $b = 0$ (e.g., the adversary receives all real outputs in his challenge), and for $j \in [1, q]$ game G^j is the same as game G^{j-1} except during the j^{th} generate call / iteration we replace the output R_j (line 19) and state variables V_j, K_j^*, (lines 15 and 17 respectively) with uniform bit strings of appropriate length. Similarly for $j \in [\overline{0}, \overline{q}]$, we let game $G^{\overline{0}}$ be identical to game Game $\text{Fwd}^{\mathcal{A},q}_{\mathsf{HMAC\text{-}DRBG}}$ with challenge bit $b = 1$ (so the adversary receives all random outputs in his challenge), and then define $G^{\overline{j}}$ to be the same as game $G^{\overline{j-1}}$ except that in the j^{th} generate call / iteration replace the state variables V_j, K_j^* on lines 15 and 17 with random bit strings. It follows that

$$\mathsf{Adv}^{\text{fwd}}_{\mathsf{HMAC\text{-}DRBG}}(\mathcal{A}, q) = |\Pr\left[G^0 \Rightarrow 1\right] - \Pr\left[G^{\overline{0}} \Rightarrow 1\right]| \leq \sum_{i=0}^{q-1} \left(|\Pr\left[G^i \Rightarrow 1\right] - \Pr\left[G^{i+1} \Rightarrow 1\right]| + |\Pr\left[G^{\overline{i+1}} \Rightarrow 1\right] - \Pr\left[G^{\overline{i}} \Rightarrow 1\right]| \right) .$$

In order bound the gap between these games, we define for each $j \in [0, q-1]$ a further set of hybrids G_k^j for $k \in [0, 9]$, in which we successively replace outputs of HMAC under different keys with random bits strings during $(j+1)^{\text{st}}$ generate call / iteration. We begin by bounding the gap between games G^j and G^{j+1} for $j \in [0, q-1]$. Fix $j \in [0, q-1]$, and let G_0^j be equivalent to Game G^j.

We next define game G_1^j which is identical to game G_0^j except we replace all PRF outputs under K_j^* — that is to say K_j^0 in line 6 of the $(j+1)^{\text{st}}$ generate call, and V_j^* on line 18 of the *previous* generate call (with the exception of $j = 0$, for which this value is not defined) —with uniform bit strings. Recall that by the definition of hybrid G^j, key K_j^* is chosen uniformly at random. Throughout this proof, we will repeatedly invoke a standard argument to bound variants of this transition; we give a detailed description in this initial case.

We claim that there exists an adversary $\mathcal{B}_1$ in the PRF distinguishing game against HMAC such that

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

To see this, notice that $\mathcal{B}_1$ can perfectly simulate the first j generate calls by random output / state variable pairs $R_i||V_i||K_0^* \leftarrow\!\!\$\ \{0,1\}^{\beta+\ell+\kappa}$ for $i = i, \ldots, j$. To simulate the $(j+1)^{\text{st}}$ generate call, $\mathcal{B}_1$ queries V_j to his RoR oracle, sets V_j^* to the returned value, and then queries $V_j^*||0\text{x}00||addin_{j+1}$ to his oracle, this time setting K_j^0 equal to the returned string. He then uses K_j^0 and V_j^* to simulate the remainder of the game for $\mathcal{A}$. At the conclusion of the game he outputs 1 if and only if $\mathcal{A}$ does.

Notice that if $\mathcal{B}_1$ is receiving real output from his oracle, then he perfectly simulated game G_j^0; otherwise he perfectly simulates a variant of the game in which HMAC is replaced by a random function, and so the gap between these games may be bounded by a reduction to the PRF security of HMAC. A standard hybrid argument, which says that if none of the inputs to a random function collide then we may replace the outputs with random bit strings, coupled with the fact that domain separation ensures these two inputs never collide, implies the result.

Next we define Game G_2^j which is identical to Game G_1^j, except the PRF outputs in lines 7, 8 are again replaced with random bit strings. By an analogous argument to that above, coupled with the fact that due to the domain separation these inputs can never collide, we may define an adversary $\mathcal{B}_2$ such that

$$|\Pr\left[G_1^j \Rightarrow 1\right] - \Pr\left[G_2^j \Rightarrow 1\right]| \leq \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{B}_2, 2) \ .$$

Next we define game G_3^j in which the $m+2$ PRF HMAC computations in lines 9, 12, 14 are replaced by uniform bit strings. By an analogous argument to that above, we may define an adversary $\mathcal{C}$ such that

$$|\Pr\left[G_2^j \Rightarrow 1\right] - \Pr\left[G_3^j \Rightarrow 1\right]| \leq \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{C}, m+2) + (m+1)^2/2^{\ell+1} \ .$$

The first term arises since an adversary in the PRF distinguishing game can simulate both games using $m+2$ queries to his RoR oracle and in time $T' \approx T$, allowing us to replace HMAC with a random function. The second term arises from using a birthday bound to upper bound the probability that two of $m+1$ randomly sampled inputs to the random function collide (again due to the domain separation of the input in line 14, this input will never cause a collision with any of the others).

In Game G_4^j, the two PRF HMAC calls on lines 15, 17 are replaced with random bit strings; again due to the domain separation the corresponding inputs can never collide and so we can define an adversary $\mathcal{B}_3$ such that

$$|\Pr\left[G_3^j \Rightarrow 1\right] - \Pr\left[G_4^j \Rightarrow 1\right]| \leq \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{B}_3, 2) \ .$$

Finally we define G_5^j which is identical to G_j^4 except now we simply sample the output $R_j \leftarrow\!\!\$\ \{0,1\}^\beta$ at random, as opposed to concatenating and truncating the random bits strings sampled during the For loop in line 11. In both cases R_j is chosen randomly and independently of all other state variables, and so state variables and so it follows that these two games are identically distributed and

$$\Pr\left[G_4^j \Rightarrow 1\right] = \Pr\left[G_5^j \Rightarrow 1\right] \ .$$

Furthermore, game G_5^j is identical to game G^{j+1}, and so putting this altogether it follows that

$$\begin{aligned} |\Pr\left[G^j \Rightarrow 1\right] - |\Pr\left[G^{j+1} \Rightarrow 1\right]| \\ \leq 3 \cdot \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{B}, 2) + \\ \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{C}, m+2) + \frac{(m+1)^2}{2^{\ell+1}} \ . \end{aligned} \quad (3)$$

With this in place, we now bound the gaps between games G^{j+1} and G^j for $j \in [0, q-1]$. We argue by a series of analogous steps to those above in reverse order, working towards returning the state variables V_j, K_j^* to being computed honestly, while preserving the random output R_j. Since the steps are similar, we describe the first more fully, and sketch the rest.

We first define a game G_6^j which is identical to game G_5^j except we replace the strings V_{j+1} and K_{j+1}^0 in lines 15, 17 of the $(j+1)^{\text{st}}$ generate call with PRF HMAC outputs.

We may then define an adversary $\mathcal{B}_1'$ in the PRF security game, who proceeds as follows. $\mathcal{B}_1'$ perfectly simulates the first j generate calls by choosing random state pairs $V_i||K_i^* \leftarrow\!\!\$\ \{0,1\}^{m+\ell+\kappa}$ for $0 = 1, \ldots, j$, and random outputs $R_i \leftarrow\!\!\$\ \{0,1\}^\beta$ for $i = 1, \ldots, q$. To simulate the $(j+1)^{\text{st}}$ generate call, $\mathcal{B}_1'$ chooses $V_j^m \leftarrow\!\!\$\ \{0,1\}^\ell$, and submits V_j^m to his RoR oracle, setting V_{j+1} to the returned string. He then submits $V_{j+1}||0\text{x}00||addin_{j+1}$ to his RoR oracle, and sets K_{j+1}^* to the returned string. He then continues to simulate the game using these state variables. By the same argument as those used above, it follows that,

$$|\Pr\left[G_5^j \Rightarrow 1\right] - \Pr\left[G_6^j \Rightarrow 1\right]| \leq \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{B}_1', 2) \ .$$

Next we define game G_7^j, which is identical to G_6^j except the randomly sampled strings in lines 9, 12, 14 are replaced with

PRF outputs. By an analogous argument that above, we can define an adversary $\mathcal{C}'$ running in time $T \approx T'$ such that

$$|\Pr\left[G_6^j \Rightarrow 1\right] - \Pr\left[G_7^j \Rightarrow 1\right]| \leq \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{C}, m+2) + \frac{(m+1)^2}{2^{\ell+1}} .$$

In G_7^j and G_8^j in which we replace the random outputs on first lines 7, 8 of the $(j+1)^{\text{st}}$ call followed by lines 6 of the $(j+1)^{\text{st}}$ call, and line 18 of the previous generate call with PRF outputs. As above, we may define adversaries $\mathcal{B}_2'$, $\mathcal{B}_3'$ running in time $T' \approx T$ such that these gaps are bounded by $\mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{B}_2', 2)$ and $\mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{B}_3', 2)$ respectively. Furthermore, notice that game G_8^j is identical to game $G^{\overline{j}}$. It follows that

$$\begin{aligned} |\Pr\left[G^{\overline{j+1}} \Rightarrow 1\right] - |\Pr\left[G^{\overline{j}} \Rightarrow 1\right]| \\ \leq 3 \cdot \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{B}', 2) + \\ \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{C}', m+2) + \frac{(m+1)^2}{2^{\ell+1}} . \end{aligned} \quad (4)$$

Putting this all together, we conclude that

$$\begin{aligned} \mathsf{Adv}^{\text{fwd}}_{\mathsf{HMAC\text{-}DRBG},\beta}(\mathcal{A}, q) \leq 6q \cdot \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{B}, 2) \\ + 2q \cdot \mathsf{Adv}^{\text{prf}}_{\mathsf{HMAC}}(\mathcal{C}, m+2) + \frac{q \cdot (m+1)^2}{2^{\ell}} . \end{aligned}$$

2) Theorem 4.3: **Proof:** We argue by a series of game hops, shown in Figure 9. All addition is understood to be modulo 2^{len} where $\mathsf{SH} : \{0,1\}^* \to \{0,1\}^\ell$.

We begin by defining G_0 which is identical to game $\text{Fwd}^{\mathcal{A},q}_{\mathsf{HASH\text{-}DRBG},\beta}$ with challenge bit $b = 0$. We also set a number of flags, although these do not affect the outcome of the game. For each $i \in [1, q]$, we let $\mathsf{coll}_i = \{V_{i-1} - m + 1, \ldots, V_{i-1} + m - 1\}$ where V_{i-1} denotes the state at the commencement of the j^{th} generate call.

Fix $j \in [1, q]$, and notice that at the end of the j^{th} generate call, the state is updated as $V_j = V_{j-1} + H_{j-1} + C + cnt_{j-1}$ where $H_{j-1} = \mathsf{SH}(0\text{x}03||V_{j-1}) \in \{0,1\}^\ell$ (see line 15 in Figure 9). Notice that if the updated state V_j is such that $V_j \in \mathsf{coll}_i$ for $i \in [1, j]$, then during the $(j+1)^{\text{st}}$ generate call the random oracle SH will be queried on a point upon which it was already queried during output generation in the i^{th} generate call. We denote the event that this occurs bad_1, and let the event that such a collision occurs before or during the state update at the conclusion of the j^{th} generate call be denoted bad_1^j.

We then define a modified game G_1 which is identical to G_0 except that if the j^{th} state update will cause bad_1^j to be set, we resample H_{j-1} as $H_{j-1} \leftarrow\!\!\$\ \{0,1\}^\ell / \mathsf{Bad\text{-}Y}_j$ where $\mathsf{Bad\text{-}Y}_j$ denotes the set of strings which would cause bad_1^j to be set; formally

$$\begin{aligned} \mathsf{Bad\text{-}Y}_j = \{Y \in \{0,1\}^\ell \mid \exists i \in [1, j] \ : \\ V_{j-1} + Y + C + cnt_{j-1} \in \mathsf{Coll}_i\} . \end{aligned}$$

After generating the challenge output, we return SH responding to all fresh queries with bit strings drawn uniformly from $\{0,1\}^\ell$. These games run identically unless the flag bad_1 is set; as such the fundamental lemma of game playing [13] implies that

$$|\Pr\left[G_0 \Rightarrow 1\right] - \Pr\left[G_1 \Rightarrow 1\right]| \leq \Pr\left[\mathsf{bad}_1 = 1 \text{ in } G_1\right] .$$

We now bound the probability of this event occurring. It follows that

$$\begin{aligned} \Pr\left[\mathsf{bad}_1 = 1\right] = \Pr\left[\mathsf{bad}_1^q = 1\right] \\ \leq \sum_{k=1}^{q} \Pr\left[\mathsf{bad}_1^k = 1 \mid \mathsf{bad}_1^{k-1} = 0\right] . \end{aligned}$$

We claim that for each $k \in [1, q]$,

$$\Pr\left[\mathsf{bad}_1^k = 1 \mid \mathsf{bad}_1^{k-1} = 0\right] \leq \frac{k(2m-1)}{2^\ell} .$$

To see this, notice that the fact that $\mathsf{bad}_1^{k-1} = 0$ means that no collisions have occurred up to and including the $(k-1)^{\text{st}}$ state update. As such, all intervals coll_i are disjoint for $i \in [1, k]$. Since each of these k intervals contains $2m-1$ distinct points, it follows that there are $k(2m-1)$ distinct values with which the new state V_k might collide and cause the flag bad_1^k to be set. Furthermore, $\mathsf{bad}_1^{k-1} = 0$ implies all V_{i-1} for $i \in [1, k]$ are distinct, and so the value $H_{k-1} = \mathsf{SH}(0\text{x}03||V_{k-1})$ used in the k^{th} state update is the result of a fresh oracle query. As such, each $H_{k-1} \in [0, 2^\ell - 1]$ is chosen with probability $2^{-\ell}$, so taking a union bound justifies the claim. It follows that

$$\begin{aligned} \Pr\left[\mathsf{bad}_1 = 1\right] \leq \sum_{k=1}^{q} \frac{k(2m-1)}{2^\ell} \\ = \frac{q(q+1)(2m-1)}{2^{\ell+1}} . \end{aligned}$$

Notice that in game G_1 oracle SH will never be queried upon the same point twice during the generation of the challenge output (due to the domain separation of the queries made in the computation of C in 5, these queries will never collide with each other or any of the other queries, and by the argument made above all queries made during output generation and state updates are distinct also). Furthermore, notice that all queries to SH contain some element in $\cup_{i=1}^{q} \mathcal{J}_i$ where we define $\mathcal{J}_i = \{V_{i-1}, V_{i-1}+1, \ldots, V_{i-1}+(m-1)\}$ where V_{i-1} denotes the state at the commencement of the i^{th} generate call.

Next we define game G_2 which is identical to G_1 except that now if the attacker queries the random oracle SH on a point upon which it was queried during the challenge generation phase, it responds with a fresh string chosen randomly from $\{0,1\}^\ell$ instead of the value previously set. It follows that these two games run identically unless $\mathcal{A}$ queries SH on a point upon which it was previously queried in the challenge generation phase, and the flag bad_2 is set. As such the fundamental lemma of game playing implies.

$$|\Pr\left[G_1 \Rightarrow 1\right] - \Pr\left[G_2 \Rightarrow 1\right]| \leq \Pr\left[\mathsf{bad}_2 = 1 \text{ in } G_2\right] .$$

Next we define game G_3, in which instead of querying the random oracle SH on the appropriate inputs during challenge generation, we simply sample uniform bit strings from the appropriate ranges. In the attacker's view these games are

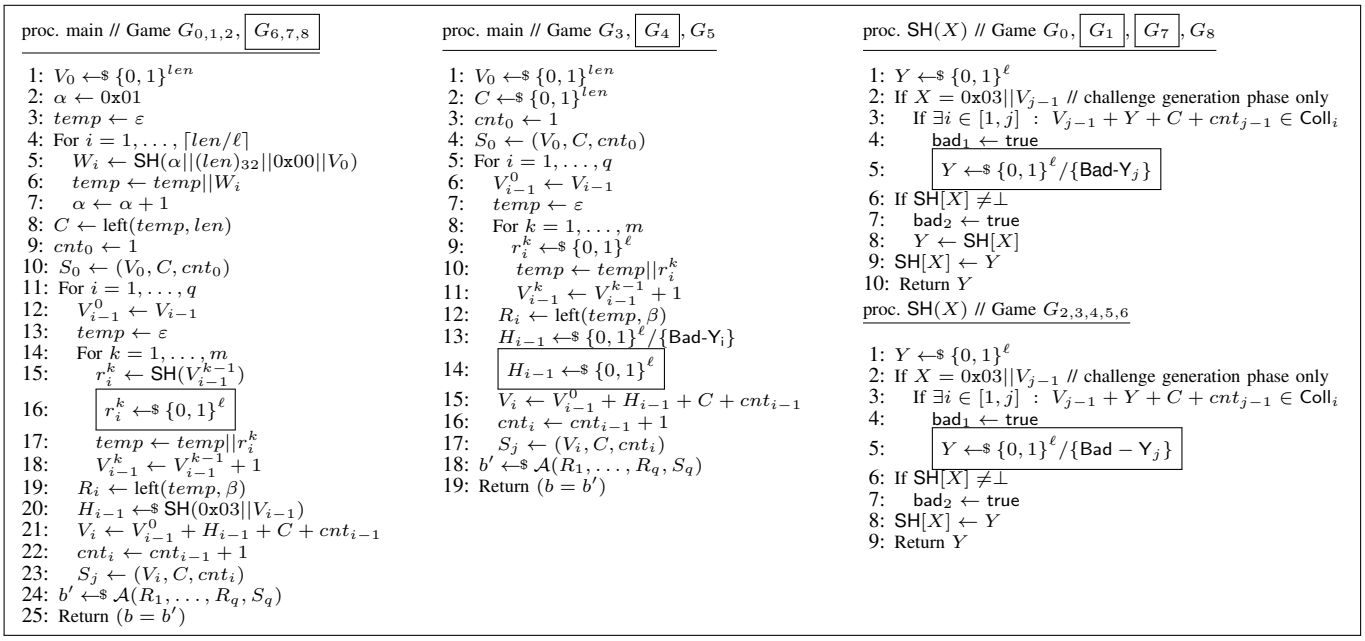

Fig. 9: Pseudocode for proof of Theorem. For each (V_{j-1}, C, cnt_{j-1}), we let $\mathsf{Bad\text{-}Y} = \{Y \mid \exists i \in [1, j-1] \ : \ V_{j-1} + Y + C + cnt_{j-1} \in \mathsf{Coll}_i\}$.

identically distributed and so

$$\Pr[G_2 \Rightarrow 1] = \Pr[G_3 \Rightarrow 1], \text{ and}$$

$$\Pr[\mathsf{bad}_2 = 1 \text{ in } G_2] = \Pr[\mathsf{bad}_2 = 1 \text{ in } G_3].$$

To bound $\Pr[\mathsf{bad}_2 = 1 \text{ in } G_3]$, we first move to game G_4 in which we sample the strings H_{j-1} used for state updates uniformly from $\{0,1\}^\ell$, instead of from $\{0,1\}^\ell/\mathsf{Bad\text{-}Y}_j$. This shall simplify bounding the probability that $\mathsf{bad}_2 = 1$ is set. By the same argument as before, it follows that

$$|\Pr[G_3 \Rightarrow 1] - \Pr[G_4 \Rightarrow 1]| \leq \frac{q(q+1)(2m-1)}{2^{\ell+1}}.$$

and

$$\Pr[\mathsf{bad}_2 = 1 \text{ in } G_3] \leq \Pr[\mathsf{bad}_2 = 1 \text{ in } G_4] + \frac{q(q+1)(2m-1)}{2^{\ell+1}}.$$

We now bound the probability that $\mathsf{bad}_2 = 1$ is set in game G_4. Recall that for each $i \in [1, q]$, the state V_i is of the form $V_{i-1} + C + H_{i-1} + cnt_{i-1}$ where $cnt_{i-1} = i$; as such we may write

$$\begin{aligned} V_i &= V_0 + i \cdot C + \sum_{j=1}^{i} j + \sum_{j=1}^{i} H_{j-1} \\ &= V_q - (q-i) \cdot C - \sum_{j=i+1}^{q} j - \sum_{j=i+1}^{q} H_{j-1}, \quad (5) \end{aligned}$$

where recall addition is modulo 2^{len}. As such while the outputs R_i which $\mathcal{A}$ is given in his challenge in game G_4 are random and independent of the previous state values, both V_q and C offer $\mathcal{A}$ some information about the previous state values which he may try to guess. Recalling that any query which may cause bad_2 to be set must contain some element in $\cup_{i=1}^{q} \mathcal{J}_i$ where $\mathcal{J}_i = \{V_{i-1}, \ldots, V_{i-1} + (m-1)\}$, taking union bounds implies that $\Pr[\mathsf{bad}_2 = 1 \text{ in } G_4]$ is less than or equal to:

$$\begin{aligned} &\Pr[\mathcal{A} \text{ guesses point in } \cup_{i=1}^{q} \mathcal{J}_i \text{ in } \sigma \text{ guesses} \mid (V_q, C)] \\ &\leq \sigma \sum_{i=1}^{q} \Pr[\mathcal{A} \text{ guesses point in } \mathcal{J}_i \text{ in 1 guess} \mid (V_q, C)] \\ &\leq \sigma \sum_{i=1}^{q} \sum_{k=0}^{m-1} \Pr[\mathcal{A} \text{ guesses } V_{i-1} + k \text{ in 1 guess} \mid (V_q, C)] \\ &\leq \sigma q m \times 2^{-\tilde{\mathrm{H}}_\infty(V_i + k | (V_q, C))}. \end{aligned}$$

We claim that for each $i \in [1, q]$ and each $k \in [0, m-1]$,

$$2^{-\tilde{\mathrm{H}}_\infty(V_{i-1} + k | (V_q, C))} \leq 2^{-\ell}.$$

To see this, recall that

$$\begin{aligned} &2^{-\tilde{\mathrm{H}}_\infty(V_{i-1} + k | (V_q, C))} \\ &\quad = \sum_{V_q, C} \max_x \Pr[V_{i-1} + k = x \mid V_q, C] \cdot \Pr[V_q, C] \end{aligned} \quad (6)$$

Now by construction in G_4, $C \leftarrow\$ \{0,1\}^{len}$, and the fact that $V_0 \leftarrow\$ \{0,1\}^{len}$ and $V_q = V_0 + q \cdot C + \sum_{i=1}^{q} (H_{i-1} + i)$ implies that $\Pr[V_q \mid C] = 2^{-len}$ also. Furthermore, for each $V_q = v_q, C = c$, it holds that

$$\max_x \Pr[V_{i-1} + k = x \mid V_q = v_q, C = c]$$

$$= \max_x \Pr\Big[V_q - (q-i)\cdot C - \sum_{j=i+1}^{q} j - \sum_{j=i+1}^{q} H_{j-1} = x-k \mid V_q = v_q, C = c\Big]$$

$$= \max_x \Pr\left[v_q - (q-i)\cdot c - \sum_{j=i+1}^{q} j - \sum_{j=i+1}^{q} H_{j-1} = x-k\right]$$

$$= \max_x \Pr\left[\sum_{j=i+1}^{q} H_{j-1} = v_q - (q-i)\cdot c - \sum_{j=i+1}^{q} j - x - k\right]$$

$$\leq 2^{-\ell} .$$

Here the first equality follows from rewriting $V_{i-1} = V_q - (q-i)\cdot C - \sum_{j=i+1}^{q} j - \sum_{j=i+1}^{q} H_{j-1}$ as per equation 5. To justify the final step, notice that since $q < 2^{48}$, and we have assumed that $\ell < len - 48$, the maximum sum of the H_{i-1} (not reduced modulo 2^{len}) is $2^{\ell+48} < 2^{len}$. (One can verify from [**?**] that the same holds for all allowed instantiations of the HASH-DRBG.) Therefore, we can treat the sum of the H_{j-1} in the above equation as an integer, rather than first reducing modulo 2^{len}. We argue by induction that for all $k \in [1, q]$,

$$\max_{z\in[0,k(2^\ell-1)]} \Pr\left[\sum_{i=1}^{j} H_{i-1} = z\right] \leq 2^{-\ell}$$

where the probability is over the choice of $H_{i-1} \leftarrow\!\!\$\ \{0,1\}^\ell$ for $i \in [1, k]$. The base case clearly holds $\Pr[\, H_0 = z\,] = 2^{-\ell}$ for each $z \in [0, 2^\ell - 1]$. We assume the hypothesis holds for $k = n$, so $\max_{z\in[0,n(2^\ell-1)]} \Pr[\, \sum_{i=1}^{n} H_{i-1} = z\,] \leq 2^{-l}$.

Now let Z^- denote the distribution of $\sum_{i=1}^{n} H_{i-1}$ where each $H_{i-1} \leftarrow\!\!\$\ \{0,1\}^\ell$. Then for $k = n+1$ (and since no wraparound occurs when adding a further term $H_n \leftarrow \{0,1\}^\ell$ to the total) it follows that for any $z \in [0, (n+1)(2^\ell - 1)]$

$$\Pr\left[\sum_{i=1}^{n+1} H_{i-1} = z\right] = \Pr\left[\, H_n = z - Z^-\,\right]$$

$$= \sum_{z^-\in[0,n(2^\ell-1)]} \Pr\left[\, H_n = z - z^-\,\right]\Pr\left[\, Z^- = z^-\,\right]$$

$$\leq \sum_{z^-\in[z-(2^\ell-1),z]} 2^{-2\ell} = 2^{-\ell} ,$$

where the final line follows from the fact that the induction hypothesis implies $\Pr[\, Z^- = z^-\,] \leq 2^{-\ell}$, and since $H_n \in [0, (2^\ell - 1)]$, it holds that $\Pr[\, H_n = z - z^-\,] = 2^{-\ell}$ if $z^- \in [z - (2^\ell - 1), z)]$ and 0 otherwise, proving the claim. Plugging this back into equation 6 yields

$$2^{-\tilde{\mathrm{H}}_\infty(V_{i-1}+k|(V_q,C))} \leq \sum_{V_q,C} 2^{-\ell}\cdot 2^{-2len}$$

$$= 2^{-l} ,$$

as required. Moving onwards, we define game G_5 which is identical to game G_4 except we return to sampling $H_{j-1} \leftarrow\!\!\$\ \{0,1\}^\ell / \mathsf{Bad\text{-}Y_j}$ during state updates. As before, it follows that

$$|\Pr[\, G_4 \Rightarrow 1\,] - \Pr[\, G_5 \Rightarrow 1\,]| \leq \frac{q(q+1)(2m-1)}{2^{\ell+1}} .$$

In game G_6, we now return to querying SH to compute the state variables, although continue to sample the outputs as uniform random bit strings. Since the random oracle is still set to return a uniform bit string even if the attacker queries it on one of these values during the guessing phases of the game, these two games are identically distributed and so

$$\Pr[\, G_5 \Rightarrow 1\,] = \Pr[\, G_6 \Rightarrow 1\,] .$$

In game G_7 we return the random oracle SH to answer honestly if the attacker queries it on a value upon which queried upon which it was queried in the guessing phase of the game. Notice that this time since SH is no longer queried to generate output, all queries to SH contain one of the state variables $\{V_0, \ldots, V_{q-1}\}$, a smaller set of possible 'bad' queries that we had previously. Therefore an analogous argument to that above implies that

$$|\Pr[\, G_6 \Rightarrow 1\,] - \Pr[\, G_7 \Rightarrow 1\,]| \leq \Pr[\, \mathsf{bad}_2 = 1 \text{ in } G_6\,]$$

$$\leq \Pr[\, \mathsf{bad}_2 = 1 \text{ in } G_4\,] + \frac{q(q+1)(2m-1)}{2^{\ell+1}}$$

$$\leq \Pr\Big[\, \mathcal{A} \text{ guesses point in } \cup_{i=0}^{q-1}\{V_i\} \text{ in } \sigma \text{ guesses} \mid (V_q, C)\,\Big] + \frac{q(q+1)(2m-1)}{2^{\ell+1}}$$

$$\leq \frac{\sigma q}{2^\ell} + \frac{q(q+1)(2m-1)}{2^{\ell+1}} .$$

Finally in game G_8 we return to oracle SH to sampling the H_i without replacement, and so again

$$|\Pr[\, G_7 \Rightarrow 1\,] - \Pr[\, G_8 \Rightarrow 1\,]| \leq \frac{q(q+1)(2m-1)}{2^{\ell+1}} .$$

Finally, notice that since in game G_8 the attacker receives random strings as outputs in conjunction with the real final state S_q, we have that G_8 is identical to game $\mathrm{Fwd}^{\mathcal{A},q}_{\mathsf{HASH\text{-}DRBG},\beta}$ with challenge bit $b = 0$. Putting this all together, we conclude that for any adversary $\mathcal{A}$ making at most σ to oracle SH, it holds that

$$\mathsf{Adv}^{\mathrm{fwd}}_{\mathsf{HASH\text{-}DRBG},\beta}(\mathcal{A}, q) \leq \frac{q(\sigma(m+1) + 3q(q+1)(m-1))}{2^\ell} .$$

∎

C. Additional Algorithms and Sample Parameters

In this section, we describe the setup and derivation function algorithms of each of the NIST DRBGs. In these descriptions we omit an optional personalization string which may be incorporated into the state during the setup process.

Algorithm 6 HMAC-DRBG setup

Require: I, $nonce$
Ensure: $S_0 = (K_0, V_0, cnt_0)$
  $seed\_material \leftarrow I||nonce$
  $K \leftarrow$ 0x00 . . . 00
  $V \leftarrow$ 0x01 . . . 01
  $(K_0, V_0) \leftarrow \mathsf{update}(seed\_material, K, V)$
  $cnt_0 \leftarrow 1$
  return (K_0, V_0, cnt_0)

The derivation function CTR-DRBG df is shown below. takes as input a string $input\_string$ (which must be a multiple of 8 bits), along with the required number of bits to be returned num_bits. The CTR-DRBG df returns an error if $num\_bits > 512$; we omit this check from the pseudocode description below for simplicity.

Algorithm 7 CTR-DRBG df

Require: : $input\_string$, num_bits
Ensure: : req_bits
  $L \leftarrow (\mathsf{len}(input\_string)/8)_{32}$
  $N \leftarrow (num\_bits/8)_{32}$
  $Z \leftarrow L||N||input_string||$0x80
  while $\mathsf{len}(Z) \bmod \ell \neq 0$ **do**
    $Z \leftarrow Z||$0x00
  $temp \leftarrow \varepsilon$
  $i = (0)_{32}$
  $K \leftarrow$ left(0x000102...1D1E1F, κ)
  while $len(temp) < \kappa + \ell$ **do**
    $IV \leftarrow (i)_{32}||0^{\ell-32}$
    $temp \leftarrow temp||\mathsf{BCC}(K, (IV||Z))$
    $i \leftarrow i + 1$
  $K \leftarrow$ left$(temp, \kappa)$
  $X \leftarrow$ select$(temp, \kappa + 1, \kappa + \ell)$
  $temp \leftarrow \varepsilon$
  while $len(temp) < num\_bits$ **do**
    $X \leftarrow \mathsf{E}(K, X)$
    $temp \leftarrow temp||X$
  $req\_bits \leftarrow$ left$(temp, num_bits)$
  return (req_bits)

Algorithm 8 BCC

Require: : $K, data$
Ensure: : $output\_block$
  $chain = 0^{\ell}$
  $n = len(data)/\ell$
  Starting with the leftmost bits of data, split $data$ into n blocks of ℓ bits each, forming B_1 to B_n
  for i = 1, …, n **do**
    $M = chain \oplus B_i$
    $chain = \mathsf{E}(K, M)$
  $output\_block \leftarrow chain$
  return $(output\_block)$

Algorithm 9 CTR-DRBG setup

Require: : I, $nonce$
Ensure: : $S_0 = (K_0, V_0, cnt_0)$
1: $seed\_material \leftarrow I||nonce$
2: **if** derivation function used **then**
3: $seed\_material \leftarrow \mathsf{df}(seed\_material, (\kappa + \ell))$
4: $K \leftarrow 0^{\kappa}$
5: $V \leftarrow 0^{\kappa}$
6: $(K_0, V_0) \leftarrow \mathsf{update}(seed\_material, K, V)$
7: $cnt_0 \leftarrow 1$
8: **return** (K_0, V_0, cnt_0)

Algorithm 10 HASH-DRBG_df

Require: : $input\_string$, $(num\_bits)_{32}$
Ensure: : req_bits
  $temp \leftarrow \varepsilon$
  $m \leftarrow \lceil num\_bits/\ell \rceil$
  $\alpha \leftarrow$ 0x01
  for $i = 1, \ldots, m$ **do**
    $temp \leftarrow temp||\mathsf{SH}(\alpha||(num\_bits)_{32}||input\_string)$
    $\alpha \leftarrow \alpha + 1$
  $req\_bits \leftarrow$ left$(temp, num_bits)$
  return req_bits

Algorithm 11 HASH-DRBG setup

Require: : $I, nonce$
Ensure: : $S_0 = (V_0, C, cnt_0)$
  $seed\_material \leftarrow I||nonce$
  $V_0 \leftarrow$ HASH-DRBG_df(seed_material, len)
  $C \leftarrow$ HASH-DRBG_df(0x00$||\mathsf{V}_0$, len)
  $cnt_0 \leftarrow 1$
  return (V_0, C, cnt_0)