Randomness in Cryptography: Theory Meets Practice

Daniel Oliver James Hutchinson

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

Information Security Group
Department of Mathematics
Royal Holloway, University of London

2018

Declaration

These doctoral studies were conducted under the supervision of Prof. Kenneth G. Paterson.

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Daniel Oliver James Hutchinson
March, 2018

Acknowledgements

I'd like to first thank my supervisor, Kenny Paterson, for his patience, help, and liberal application of boot to my rear.

I'm exceptionally grateful to my parents, Wendy and Chris; I'm so glad you got to see me finish this and that I made you proud.

I'd like to thank my sister, Claire and the rest of my family, for their love and support.

Thanks to my friends, old and new, there are many of you to list, a list too large for this margin.

Special thanks goes to Demelza; without your love and support the thesis would've beaten me.

Thank you to: James, Lotta, Chris, Ellie; my colleagues in LINKS; my officemates Chel, Christian, James, Rob and Naomi; Lindsay; my Southampton friends; Thalia, Sam, and, lastly, Aaron.

Finally, thank you to Rob Esswood for inspiring me so long ago to pursue Mathematics; it's your fault.

Abstract

Randomness is a key ingredient in every area of cryptography; and as the quote goes, producing it should not be left to chance. Unfortunately it's very difficult to produce true randomness, and consuming applications often call for large, high quality amounts on boot or in quick succession. To meet this requirement we make use of Pseudo-Random Number Generators (PRNGs) which we initialise with a small amount of randomness to produce what we hope to be high quality pseudo-random output.

In this thesis we investigate some of the different security models associated with capturing what makes a "good" PRNG, along with the problem of constructing a secure PRNG by adapting primitives available. We focus mainly on the sponge construction, noting that the original formulation does not lend itself well to a secure PRNG but with some adjustment can be made into a robust and secure PRNG. This is done by utilising a feed-forward of the inner, secure part of the sponge state, which establishes an efficient forward security mechanism.

We then present an updated security model for PRNGs designed to capture variable output subroutines present in some PRNGs where an adversary is allowed to request differing amounts of output with each call to the PRNG. We maintain the ability to prove robustness via two simpler security notions which are now extended to variable-output versions.

We then follow with an analysis of the NIST PRNGs in this new security model, which served as motivation for updating the security model. We show that under certain assumptions the NIST generators do satisfy security in this model.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ctr_drbg	The NIST DRBG based on a block cipher.
DRBG	Deterministic Random Bit Generator.
hash_drbg	The NIST DRBG based on a hash function.
PRF	Pseudo-random function.
PRG	Pseudo-Random Number Generator.
PRNG	Pseudo-Random Number Generator.
PRP	Pseudo-random permutation.
PWI	PRNG with Input.
Reverie	Our PRNG design.
sponge.prng	The SpongePRNG design.
VOPWI	Variable-Output PRNG with Input.

Notation

$x \leftarrow y$	When specifying algorithms, this denotes writing the value y to the variable x.
$x \overset{\$}{\leftarrow} y$	When $x \in X$ and X is a finite set, this denotes choosing an element of X uniformly at random and writing it to the variable x.
$x \leftarrow A(x_1, \ldots, x_n)$	For a deterministic algorithm A, running A with inputs $x_1, \ldots, x_n$ and output x.
$x \overset{\$}{\leftarrow} A(x_1, \ldots, x_n)$	For a probabilistic algorithm A, running A with inputs $x_1, \ldots, x_n$ and output x.
b	The width of a sponge state, in bits.
c	The capacity of a sponge state, in bits.
$\mathcal{D}$	A Distribution Sampler, used in security proofs to model the entropy pool available to a PRNG.
d	An enumeration of the number of entropy inputs into a PRNG.
$\mathsf{E}_k(v)$	The block cipher encryption of an input v under a key k.
$\mathsf{func_n}$	The set of all functions on n-bit strings.
H	A hash function.
$\mathbf{H}_\infty()$	Minimum entropy.
I	Entropy input into a PRNG, often from an entropy source.
λ	The security parameter of a construction.
M	A masking function which outputs an ideal state when applied to a PRNG state.
n_{E}	The size of the block cipher's input and output block.
$\mathcal{P}_n$	The set of all permutations on n-bit strings.
r	The output length of a sponge function, in bits.

$\mathcal{RO}$	A Random Oracle.
s_i	The ith state of a PRNG.
$\bar{s}_i$	The ith outer state of a sponge-based PRNG.
$\widehat{s}_i$	The ith inner state of a sponge-based PRNG.
SD	The statistical difference of two distributions.
t_i	The image of s_i under the permuation π.
$\bar{t}_i$	The outer part of the image of s_i under the permuation π.
$\widehat{t}_i$	The inner part of the image of s_i under the permuation π.
v_i	A sub-state of the ith working state of the NIST Deterministic Random Bit Generators (DRBG s).
z_i	The updated state derived from the s_i under the permuation π.
$\bar{z}_i$	The outer part of the updated state derived from the s_i under the permuation π.
$\widehat{z}_i$	The inner part of the updated state derived from the s_i under the permuation π.

Chapter 1

Introduction

Contents

This chapter gives an overview of the thesis. This includes both the motivation and contributions included in the thesis. In this chapter we also give a brief roadmap of the overall structure of the thesis.

1.1 Motivation

Research over the last few decades on cryptography has seen a divergence in theoretical and practical cryptography. The current climate of real world applications calls for more theoretically informed practical designs, along with an interest in designs that are backed by a proof in the theoretical setting. Many designs in the literature are still focused entirely on the theoretical setting with little thought of the practical restraints and considerations. This can be attributed to modelling the adversary; the strength and abilities she entails along with the environment a system is running in. Often, PRNGs are designed either entirely from the practical perspective with little involvement of theoretical-methods and literature, and are later analysed when the research has caught up. This can result in catastrophic cases of bad randomness undermining otherwise secure cryptosystems when vulnerabilities and flaws are discovered, such as papers attacking RC4 in TLS (such as [32]). Fortunately, there are an increasing number of competition-based design programs, such as the SHA3, CAESAR and NIST PQC competitions, that have meant more time and minds are focused on confirming designs before they are utilised. However, this still often draws primarily on practical considerations over theoretical, model based analysis.

We would like to further the effort to bring theoretical security closer to practical security by improving the PRNG security models, specifically those based upon [29], by using more considerations used in practice to inform the model, such as reliability of the PRNG at boot. Currently one of the largest disparities is the notion of entropy estimation in a generator. In practice, a generator must utilise a mechanism to estimate the current entropy of the state of the generator, which is often realised by using health checking systems that monitor both ingoing entropic strings and the output of the generator. In theoretical security models this is avoided by making it part of the adversary to guarantee minimum entropy entering into the PRNG, while keeping the possible entropy sources as wide as possible; in practice, entropy sources may be far more reliable than is assumed in theory. This is obviously beneficial in that a theoretically secure generator secure against relatively low or unreliable entropy sources paired with a "better" entropy source should perform even better in practice.

Another major difference between theory and practice in the context of PRNGs is the notion of the seed of a generator. In practice, the generator must initialise itself with what entropy is available, which may cause issues on its own when this initial entropy is insufficient. We will touch on this below. In the theoretical setting we often rely on a uniformly random seed generated at initialisation by the generator from an independent entropy source to mitigate a class of attacker, or rather entropy source, that can inform the future states of a generator. In practice this is not the case, since there is not enough entropy and, as we stated above, the class of entropy sources is much more restricted in the amount of output available, though generally at a higher quality than is assumed in theory.

We now provide a brief explanation of what problems can arise from bad randomness and how it is more than just a theoretical problem. The authors of [37] and [41] investigated the security of certificates produced by factory-installed PRNGs and find some worrying conclusions.

In [37] the authors used a three-phase data collection process consisting of:

1. Host discovery: Scanning the public IPv4 address space to find hosts with port 443 (HTTPS) or port 22 (SSH) open.

2. Certificate and host-key retrieval: The second phase was to attempt either a TLS or SSH handshake and storing the presented certificate chain for TLS or the host-key for SSH.

3. Lastly the TLS certificate chains were parsed to generate a relational database from the X.509 fields.

This led to 5.8 million unique TLS certificates from 12.8 million hosts and 6.2 million unique SSH host keys from 10.2 million hosts. Of these, 5.57% of TLS hosts and 9.60% of SSH hosts used the same keys as other hosts in an apparently vulnerable way. The authors were in fact able to compute the private keys for 64,000 (or 0.50%) of the TLS hosts and 108,000 (1.06%) of the SSH hosts with no other information by exploiting weaknesses in RSA and DSA when used with insufficient randomness.

The authors use an interesting quasilinear greatest common divisor (gcd) finding algorithm to compute the pairwise gcds of all the RSA moduli, yielding 66,540 vulnerable hosts who shared one of their RSA prime factors with another host in the survey. Vulnerable hosts included routers, server management cards, firewalls and other network devices, all of which are headless[1] and lack a human input in terms of random entropy or, in other words, have access to only limited entropy sources.

Throughout the paper the authors have tried to determine exactly why they have found what they found and have done more research to find causes of confusion, such as a collection of moduli based on all combinations of nine primes which turned out to be a single company.

In [41] the authors collected 11.7 million public keys using a variety of sources, including the EFF SSL repository. They collected 6.4 million distinct RSA moduli with the remaining keys roughly evenly split between ElGamal keys and DSA keys, plus a single ECDSA key. They found that of the 6.4 million RSA moduli, 4% shared their RSA modulus, often involving unrelated parties. They also found that 1.1% of the RSA moduli occur more than once, some of them thousands of times. They determined that 12,934 RSA moduli offered no security, either due to expired

[1]Headless means with no direct input or user interface (UI) output such as a monitor.

certificates or use of MD5 as a hash function. However 5250 of the certificates, including 3201 distinct 1024-bit RSA moduli are still valid and use SHA1.

Interestingly the authors went on to model the RSA moduli as graphs with edges corresponding to the moduli connecting two (hopefully) unidentifiable primes as vertices. Ideally, for c distinct moduli it is hoped that the graph representation consists of c connected components, of one edge connecting two unknown primes. Unfortunately, even for a small amount of moduli this was not the case; (see [41, page 7-8]).

The authors concluded that the amount of shared primes, especially RSA primes, is unacceptable and worrying, especially since many are still used and valid. The authors have suggested that this is possibly due to incorrect seeding of random number generators. They go on to show support for NIST's decision to adopt DSA as a digital signature standard but point out the weakness of ElGamal and ECDSA if the required random nonce is not properly used.

These issues are not restricted to just the generation of RSA keys. In December 2010 there was the Sony Playstation hack where the ECDSA key was recovered due to incorrect random nonce generation. In August 2013 there was a bug in android that meant the PRNG was not initialised correctly and resulted in a similar recovery of ECDSA keys that in turn, led to the theft of large sums of bitcoins. Similarly, an HTTP session ID compromise via the java servlets engine was published in [35]. In the 2010 paper by Ristenpart and Yilek [52], the authors reuse of random values via reusing a VM snapshot lead to vulnerabilities in TLS such as compromise of Diffie Hellman key exchange. A talk at the Blackhat USA conference [10] by Becherer, Stamos and Wilcox presented how bad randomness can be used in an attack in the cloud setting, such as requesting large numbers of password resets for a single account, which all remain valid and active, followed by attempting to guess one of these randomly generated password reset urls to gain access to an account.

1.2 Contributions

We begin addressing some of the shortcomings mentioned in Section 1.1 in this thesis. Namely, we present an improved PRNG design with practicality in mind, followed by extending the security model to better capture practical requirements of PRNGs. Finally, we provide a security analysis of a pair of NIST PRNGs that, at the time of writing, have not received any formal analysis. The main contributions of this thesis are summarised below:

- We provide an improved sponge-like PRNG design and prove that it satisfies the robustness security property in the ideal permutation model, where the underlying permutation is modelled as a public, random permutation. Our design requires far fewer calls to the underlying permutation which has efficiency benefits, especially in constrained environments and makes collisions of the state of the generator less likely. We prove that although this design departs from the sponge design, it is still encapsulated by the more general family of parazoa designs. We present some implementation efficiency comparisons between our design and the original sponge-based design.
- We next extend the current security model, allowing for analysis of PRNGs with subroutines to produce variable amounts of output without updating the internal state of the generator; or, in other words, PRNGs that allow an adversary to request varying amounts of output with each call to the generator. We also add several changes that we believe allows us to better specify and model PRNGs in practise. We define the notion of a variable-output PRNG with input, or, VOPWI which allows an adversary to vary the amount of output received per internal state update of the generator. This new notion allows us to analyse the security of PRNGs when outputting large amounts of output quickly versus a smaller amount over many more calls. We formalise the need for a seedgen algorithm and how the setup algorithm should be provided with entropy to generate the initial state of a generator.

- Using our updated security model we analyse the NIST PRNGs, namely the hash_drbg and ctr_drbg. We analyse both designs in our new VOPWI security model, while discussing the different design choices and how they affect analysis. We then prove the generators are robust under several necessary assumptions.

Parts of the research findings in this thesis have been published at SAC 2016 [39], namely, Chapter 3.

1.3 Structure of Thesis

The main body of the thesis is organised as follows:

Chapter 2 In this chapter we provide the preliminaries and background necessary for the rest of the thesis. Some sections will refer back to parts of this chapter for basic definitions, while more complex or altered definitions are included in context throughout the remaining chapters.

Chapter 3 This chapter presents an improved design of a sponge-like PRNG based upon the original sponge PRNG design in [18]. We show that our design is more efficient due to fewer calls to the underlying permutation and prove that it satisfies the notions of preserving and recovering security. This allows us to conclude that the generator satisfies the robustness notion of security. We conclude by briefly comparing the design to the original sponge design by considering the design as part of the wider parazoa family.

Chapter 4 This chapter outlines several updates to the security model of PWIs given in [29], incorporating and building upon updates from [55]. We provide updates that allow the capture of a more practical design of a PWI, along with the ability of an adversary to request varying amounts of output per call to the next function of the generator. The latter is an almost mirror of the update the authors of [29] made to the security notion of refreshing the generator from [4] to enable an adversary to slowly feed entropy into the generator, as opposed to all at once. Our update in essence allows an adversary to extract output in small parts or all at once. We build upon the notion of a masking function by allowing different distributions for different situations in the security games, reflecting on how some parts of the state may remain unchanged or may change in different ways. We formally define new versions of preserving and recovering security before proving our own updated version of robustness.

Chapter 5 We utilise our updated security model from the previous chapter to analyse the security of the NIST random bit generators as defined in [7]. We chose to update the security model to allow us to capture the subroutines and behaviour of the NIST generators that allow for varying amounts of output at each call. Since the generators are specified in full, we are also able to formally define the other algorithms such as the seedgen algorithm, though we are forced to make an assumption on its output to achieve security. We conclude that under several strong assumptions, the generators are variable-output robust, however these assumptions do not necessarily transfer to practical implementations.

Chapter 6 We provide our conclusions and thoughts on future work in the related areas.

Chapter 2

Background Materials

Contents

This chapter introduces the notation and provides an overview of the necessary background material required for the future chapters. This includes introductory definitions of pseudo-random number generators, related security models, other useful primitives and some proof techniques that will be used.

2.1 Preliminaries

Much of this chapter is based on definitions from Katz and Lindell [40], Bellare and Rogaway [14] and the in-progress book by Boneh and Shoup [21]. For the reader's convenience, an acronym and notation list are given on Page 12.

Definition 2.1.1. Let f and g be functions on the real numbers. We say that $f(x) = O(g(x))$ as $x \to \infty$ if and only if $\exists M \in \mathbb{N}$ and $x_0 \in \mathbb{R}$ such that $|f(x)| \leq M|g(x)|$ for all $x \geq x_0$.

Definition 2.1.2. An algorithm taking input of size $k \in \mathbb{N}$ is said to be polynomial time if it always terminates in time $O(k^c)$, for some constant $c \in \mathbb{N}_0$.

Definition 2.1.3. The statistical distance between two discrete random variables X and Y over the set $\mathcal{X}$ is denoted

$$\mathsf{SD}(X, Y) = \frac{1}{2} \sum_{x \in \mathcal{X}} |\Pr[X = x] - \Pr[Y = x]| \leq \epsilon.$$

We say that X and Y are ϵ-close.

Definition 2.1.4. The minimum entropy of a random variable X is defined as

$$H_\infty(X) = \min_{x \overset{\$}{\leftarrow} X} \{-\log(\Pr[X = x])\}.$$

Definition 2.1.5. A source S^π is defined as an input-less randomised oracle which makes queries to π and outputs a string. The range of the source is denoted $[S]$ and is the set of all values the source outputs with positive probability, taken over the choice of π and the internal randomness of S.

2.1.1 Provable Security

Unless stated otherwise, we use the usual game-based formalism from [15]; for a game $\mathbf{G}, \mathbf{G}(\mathcal{A}) \Rightarrow 1$ or $\mathbf{G}^{\mathcal{A}} \Rightarrow 1$ denotes the event that an adversary $\mathcal{A}$ playing the game $\mathbf{G}$, results in the *game* outputting 1. We use $\mathbf{G}(\mathcal{A}) \rightarrow 1$ or $\mathcal{A}^{\mathbf{G}} \Rightarrow 1$ to denote the event that the *adversary* $\mathcal{A}$ playing the game $\mathbf{G}$ outputs 1. A game consists of at least two procedures, Initialise and Finalise. The Initialise procedure generally assigns initial values to variables, and passes them to the adversary or other procedures. The Finalise algorithm runs once all queries by the adversary have been made, usually followed by the adversary outputting a value. The Finalise procedure then outputs its own value which is taken to be the output of the game. A security game that does not make use of a random oracle is said to be in the "standard model"; these games will usually show equivalence to a suitably "hard" problem, such as factorisation large moduli.

Definition 2.1.6. A function $\mathsf{negl} : \mathbb{N} \rightarrow \mathbb{R}$ is called negligible if for all $c \in \mathbb{R}_{>0}$ there exists an $n_0 \in \mathbb{N}$ such that for all integers $n \geq n_0$, we have $|\mathsf{negl}(n)| \leq \frac{1}{n^c}$.

2.1.2 Pseudorandom Permutations (PRPs)

Pseudo-random permutations (PRP) take as input a key and an input string. The key picks a permutation from a family which is applied to the input. As the name implies, a PRP should be indistinguishable from a random permutation.

Definition 2.1.7. Let $P : \{0,1\}^{n_k} \times \{0,1\}^n \longrightarrow \{0,1\}^n$ be an efficient, keyed permutation. Then P is a pseudo-random permutation (PRP) family, if, for all probabilistic polynomial-time distinguishers D, there exists a negligible function negl such that:

$$\left|\Pr\left[\mathsf{D}^{P_k(\cdot)}(1^n) \implies 1\right] - \Pr\left[\mathsf{D}^{\pi(\cdot)}(1^n) \implies 1\right]\right| \leq \mathsf{negl}(n_k),$$

where the first probability is taken over the uniform distribution of $k \in \{0,1\}^{n_k}$ and the randomness of D, and the second probability is taken over the uniform distribution of $\pi \in \mathcal{P}_n$, and the randomness of D.

We sometimes refer to the family of PRPs $\{P_k : \{0,1\}^n \longrightarrow \{0,1\}^n\}$.

Definition 2.1.8. Let $P : \{0,1\}^{n_k} \times \{0,1\}^n \longrightarrow \{0,1\}^n$ be an efficient, keyed permutation. Then P is a strong pseudorandom permutation if, for all probabilistic polynomial-time distinguishers D, there exists a negligible function negl such that:

$$\left|\Pr\left[\mathsf{D}^{P_k(\cdot),P_k^{-1}(\cdot)}(1^n) \implies 1\right] - \Pr\left[\mathsf{D}^{\pi(\cdot),\pi^{-1}(\cdot)}(1^n) \implies 1\right]\right| \leq \mathsf{negl}(n),$$

where the first probability is taken over the uniform distribution of $k \in \{0,1\}^{n_k}$ and the randomness of D, and the second probability is taken over the uniform distribution of $\pi \in \mathcal{P}_n$, and the randomness of D.

2.1.3 Pseudorandom Functions (PRFs)

Here we define a generalisation of a PRP, namely a pseudo-random function or PRF. A PRF, similarly to a PRP, takes as input a key and a string. The key picks a function from a family which is applied to the input. A PRF should be indistinguishable from a random function.

Definition 2.1.9. Let $F : \{0,1\}^{n_k} \times \{0,1\}^n \longrightarrow \{0,1\}^n$ be an efficient, keyed function. We call F a Pseudo-random function (PRF) family if, for all probabilistic polynomial-time distinguishers D, there is a negligible function negl such that:

$$\left|\Pr\left[\mathsf{D}^{F_k(\cdot)}(1^n) \implies 1\right] - \Pr\left[\mathsf{D}^{f(\cdot)}(1^n) \implies 1\right]\right| \leq \mathsf{negl}(n),$$

where the first probability is taken over the uniform distribution of $k \in \{0,1\}^{n_k}$ and the randomness of D, and the second probability is taken over the uniform distribution of $f \in \mathsf{Func_n}$, and the randomness of D, where $\mathsf{Func_n}$ is the set of all functions mapping n-bit strings to n-bit strings.

We sometimes refer to the family of PRFs $\{F_k : \{0,1\}^n \longrightarrow \{0,1\}^n\}$.

2.1.4 Block Ciphers

A block cipher is a deterministic algorithm taking a key and a fixed-length input or block. A block cipher consists of two algorithms, an encryption algorithm and a decryption algorithm such that the decryption algorithm is the inverse of the encryption algorithm when queried with the same key..

```
Procedure: owf
$x \xleftarrow{\$} \{0,1\}^m$
$y \leftarrow f(x)$
$x' \leftarrow \mathcal{A}_{\mathsf{owf}}(1^\lambda, y)$
if $f(x') = y$
  return 1
else
  return 0
```

Figure 2.1: The one-way function security procedure.

Definition 2.1.10. Let $n, n_k \in \mathbb{N}$. Then an n-bit block cipher is a function $\mathsf{E} : \{0,1\}^{n_k} \times \{0,1\}^n \longrightarrow \{0,1\}^n$, where $\{0,1\}^{n_k}$ is the key space and for any $k \in \{0,1\}^{n_k}, \mathsf{E}(k, \cdot)$ is a permutation. It is common to write $\mathsf{E}_k(\cdot)$ instead of $\mathsf{E}(k, \cdot)$.

Definition 2.1.11. Define the PRP security of a block cipher $\mathsf{E} : \{0,1\}^{n_k} \times \{0,1\}^n \longrightarrow \{0,1\}^n$ as the advantage of an adversary $\mathcal{A}_{\mathsf{PRP}}$, defined as

$$\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E}}(\mathcal{A}_{\mathsf{PRP}}) = \left|\Pr\left[k \xleftarrow{\$} \{0,1\}^{n_k} : \mathcal{A}^{\mathsf{E}_k}_{\mathsf{PRP}} \implies 1\right] - \Pr\left[\pi \xleftarrow{\$} \mathcal{P}_n : \mathcal{A}^{\pi}_{\mathsf{PRP}} \implies 1\right]\right|.$$

Definition 2.1.12. Define the strong PRP security of a block cipher $\mathsf{E} : \{0,1\}^{n_k} \times \{0,1\}^n \longrightarrow \{0,1\}^n$ as the advantage of an adversary $\mathcal{A}_{\mathsf{PRP}}$, defined as

$$\mathsf{Adv}^{\pm\mathsf{PRP}}_{\mathsf{E}}(\mathcal{A}_{\mathsf{PRP}}) = \left|\Pr\left[k \xleftarrow{\$} \{0,1\}^{n_k} : \mathcal{A}^{\mathsf{E}_k, \mathsf{E}_k^{-1}}_{\mathsf{PRP}} \implies 1\right] - \Pr\left[\pi \xleftarrow{\$} \mathcal{P}_n : \mathcal{A}^{\pi, -\pi}_{\mathsf{PRP}} \implies 1\right]\right|.$$

2.1.5 Extractors and One-Way Functions (owfs)

Definition 2.1.13 (One-Way Function (owf))**.** A function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ is one-way if the following two conditions hold:

- There exists a polynomial-time algorithm M_f computing f; that is $\forall x, M_f(x) = f(x)$;
- For every probabilistic polynomial-time adversary $\mathcal{A}_{\mathsf{owf}}$, there is a negligible function negl such that

$$\Pr\left[\mathsf{owf}_f^{\mathcal{A}_{\mathsf{owf}}}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda),$$

for owf as described in Figure 2.1.

Definition 2.1.14 (Extractor from [46]). A function $\mathsf{E} : \{0,1\}^n \times \{0,1\}^d \longrightarrow \{0,1\}^m$ is a (k, ϵ)-extractor if for every distribution X over $\{0,1\}^n$ with $H_\infty(X) \geq k$, $E(X, \mathsf{U}_d)$ is ϵ-close to uniform.

2.1.6 Random Oracle Model (ROM)

We briefly outline the random oracle model, first formalised in [13]. A random oracle informally returns a uniformly random string for every unique input (of arbitrary length). We use the notation $\mathcal{RO}(x)$ to denote a random oracle taking input x. Unless stated otherwise, the random oracle will always output a fixed length output. The random oracle keeps track of previous queries and will always output the same value for previously defined input. To summarise, when the $\mathcal{RO}$ is queried on the string x, it first checks to see if the input string x has been queried before, if so it returns the value it has linked to that input. If the x has not previously been queried to $\mathcal{RO}$ then $\mathcal{RO}$ returns a uniformly random string.

2.1.7 Ideal Permutation Model (IPM)

An implementation of a PRNG may make use of a publicly available permutation. To model this it is common to use the ideal permutation model (IPM).

Formally, each party has oracle access to a public, random permutation $\pi \xleftarrow{\$} \mathcal{P}_n$, chosen by the challenger at the beginning of a game. The permutation can be queried as both π and π^{-1} but, for simplicity, we write that an algorithm or entity, such as an adversary $\mathcal{A}$, has access to π by $\mathcal{A}^\pi$. We make use of the following, which denotes the advantage of an adversary $\mathcal{A}$ with oracle access to π in distinguishing between the distributions $\mathsf{D}_0, \mathsf{D}_1$ that also have access to π:

$$\mathsf{Adv}^{\mathsf{dist}}_{\mathcal{A}}(\mathsf{D_0}^\pi, \mathsf{D_1}^\pi) = \left| \Pr\left[X \xleftarrow{\$} \mathsf{D_0}^\pi : \mathcal{A}^\pi(X) \Rightarrow 1\right] - \Pr\left[X \xleftarrow{\$} \mathsf{D_1}^\pi : \mathcal{A}^\pi(X) \Rightarrow 1\right]\right|,$$

with $\mathcal{A}$ being called a q_π-query adversary if it asks at most q_π queries to π.

2.1.8 Ideal-Cipher Model (ICM)

The ideal-cipher model (ICM) is a generalisation of the IPM (or alternatively the IPM can be seen as a case of the ICM with a public fixed choice of key). Formally, at the beginning of the game, the challenger chooses a random permutation $\pi_k \in \mathcal{P}_n$ for each key $k \in \{0,1\}^{n_k}$. We make use of the following, which denotes the advantage of an adversary $\mathcal{A}$ with oracle access to π_k in distinguishing between the distributions $\mathsf{D}_0, \mathsf{D}_1$ that also have access to π_k:

$$\mathsf{Adv}^{\mathsf{dist}}_{\mathcal{A}}({\mathsf{D}_0}^{\pi_k}, {\mathsf{D}_1}^{\pi_k}) = \left|\Pr\left[X \xleftarrow{\$} {\mathsf{D}_0}^{\pi_k} : \mathcal{A}^{\pi_k}(X) \Rightarrow 1\right] - \Pr\left[X \xleftarrow{\$} {\mathsf{D}_1}^{\pi_k} : \mathcal{A}^{\pi_k}(X) \Rightarrow 1\right]\right|,$$

with $\mathcal{A}$ being called a q_π-query adversary if it asks at most q_π queries to π_k.

2.2 Definitions of Pseudo-Random Number Generators

The main topic in this thesis is the investigation of pseudo-random number generators. In this section we will take a look at how the modelling of a PRG has evolved over time, along with how security models have changed to reflect real world situations, addressed concerns and, where there is still a large gap between theory and practice.

There are many different definitions and terms for a pseudo-random number generator; in this thesis, a PRG will refer to a Pseudo-Random Number Generator *without* input for simplicity. A PRG is the most basic definition of a pseudo-random number generator, while a PRNG or PWI will refer to a Pseudo-Random Number Generator *with* input.

2.2.1 PRGs

One of the simplest definitions of a random number generator is a PRG. Taking no additional input other than an initial state, it expands this random input into larger amounts of pseudo-random output.

Definition 2.2.1 (PRG)**.** For $n, \ell \in \mathbb{N}$, a function $\mathsf{G} : \{0,1\}^n \longrightarrow \{0,1\}^\ell$ is a deterministic (t, ϵ)-*pseudo-random number generator* (PRG) if

1. $\ell > n$, or in other words, stretches the original seed of randomness into a larger amount of pseudo-randomness.
2. No adversary running in time t can distinguish between

   $$\mathsf{G}(s_0) \text{ and } \mathrm{R} \overset{\$}{\leftarrow} \{0,1\}^\ell,$$

   for some $s_0 \overset{\$}{\leftarrow} \{0,1\}^n$ with probability greater than ϵ.

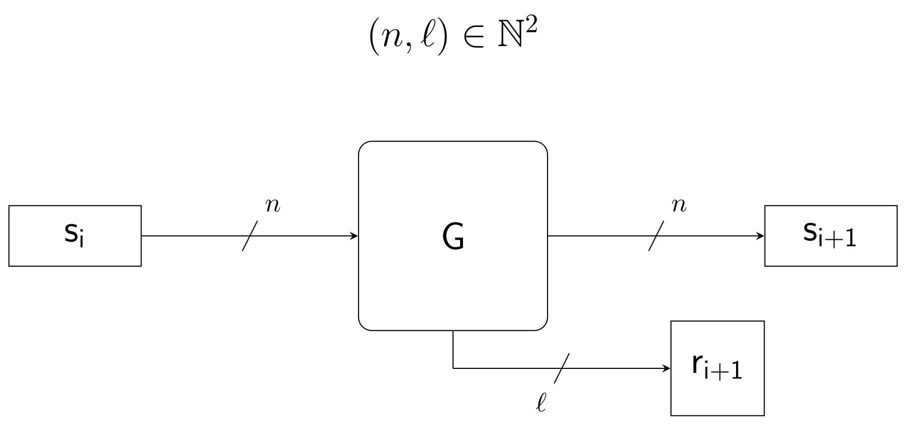


Figure 2.2: A simple PRG description.

2.2.2 PRNGs and PWIs

Definition 2.2.2 (PRNG from [29]). A PRNG with input (PWI) is a triple of algorithms $\mathsf{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ and a triple $(n, \ell, p) \in \mathbb{N}^3$ where n is the state length, ℓ is the output length, p is the input length of G and

- setup: is a probabilistic algorithm that outputs some public parameters seed for the generator, along with the initial state s_0;
- refresh: is a deterministic algorithm that, given seed, a state $s_i \in \{0,1\}^n$ and an input $\mathrm{I} \in \{0,1\}^p$, outputs a new state $s_{i+1} := \mathsf{refresh}(\mathsf{seed}, s_i, \mathrm{I})$;
- next: is a deterministic algorithm that, given seed and a state $s_i \in \{0,1\}^n$, outputs a pair $(s_{i+1}, r_{i+1}) = \mathsf{next}(s_i) = \mathsf{next}(\mathsf{seed}, s_i)$, where s_{i+1} is the new state and $r_{i+1} \in \{0,1\}^\ell$ is the output.

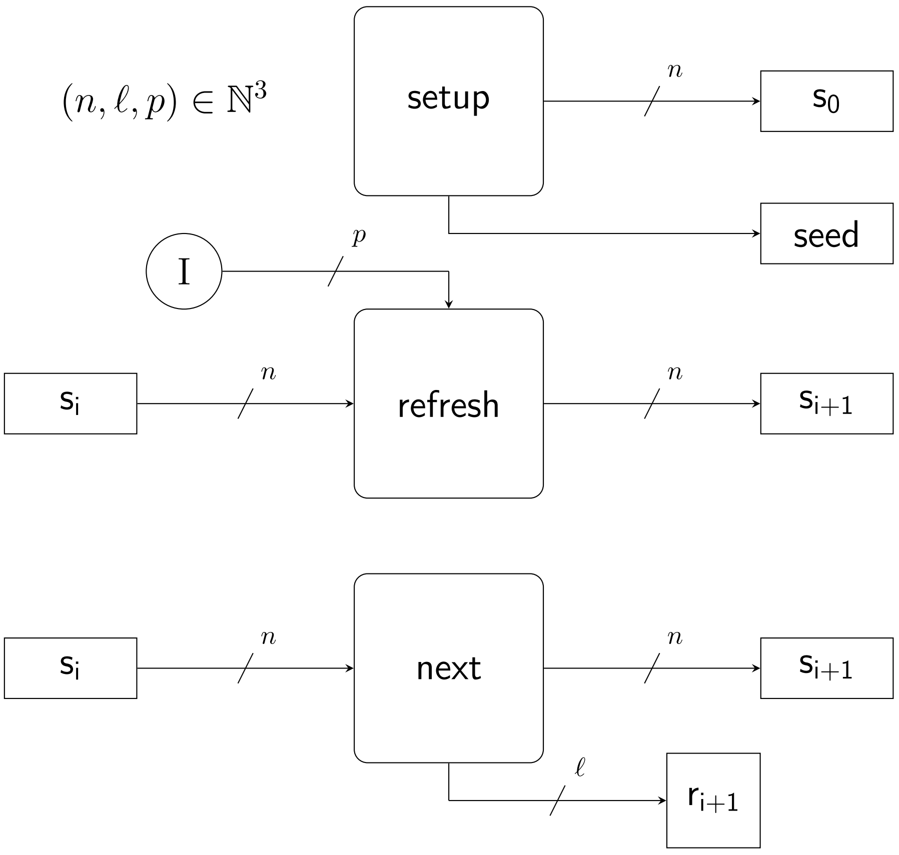


Figure 2.3: A simple PRNG description.

2.3 Existing Security Models of PRNGs

In this section we briefly survey the existing security models for pseudo-random number generators. The study of PRNGs stretches back quite far, from the early works by Blum and Micali [20], Yao [57], to work by Desai, Hevia and Yin [27] who model a PRNG as a pair of stateful algorithms (key,next), the model by Barak, Shaltiel and Tromer [6] which models the generator as (setup,next), building towards the model by Barak and Halevi [4] who use (refresh,next), to the extension of [4] by Dodis et. al. [29] which we base the work of this thesis on. We also touch upon some updated models that build upon [29], such as [30]. For a more complete list and summary, see [53]. We will begin by taking a more in-depth look at [29] and [30] which were the main influences of this thesis and incorporate the majority captured in previous models.

2.3.1 Two papers: /dev/random is not Robust and How to Eat Your Entropy and Have It Too

The security model introduced in [29] identified and incorporated the very applicable situation where a PRNG may accumulate entropy at a slow rate (through low entropy inputs) and is at risk of “prematurely” being called before enough entropy has been gathered, and focused on building upon [4]. The updated model [30] takes a less conservative approach to this situation through pre-mature get-next security. Both of these papers sought to formalise these situations and to try to match the constraints and challenges faced when instantiating a PRNG in practise with more practically inspired restraints and security notions. Even the most recent model still, by the authors’ admission, does not encompass the full desirable picture of what a secure PRNG should be, less even what a secure PRNG would look like. The model as it stands does not incorporate the entropy estimation or mixing functions present in many real world PRNGs, for instance, but does define the notion of a scheduler (which handles several entropy pools feeding into the PRNG) as used in the Fortuna PRNG [31, Chapter 9].

First we define the entropy source, which in this model is modelled as a second adversary $\mathcal{D}$ and called the distribution sampler, which does not communicate with the main adversary $\mathcal{A}$. Like all adversaries, restrictions need to be made, though these are as few as possible. The reasoning behind the entropy source being an adversary is to emulate the potentially adversarial environment where the PRNG is forced to operate. For the PRNG to be deemed in a "secure" state, the entropy of the state is required to be over a given threshold γ^* specified as part of the security of the generator.

Definition 2.3.1. The distribution sampler $\mathcal{D}$ is a stateful and probabilistic algorithm which given the current state σ outputs a tuple $(\sigma', \mathrm{I}, \gamma, z)$ where:

- σ' is the new state for $\mathcal{D}$;
- $\mathrm{I} \in \{0,1\}^p$ is the next input for the refresh algorithm;
- γ is some fresh entropy estimation of I;
- z is the leakage about I given to the adversary $\mathcal{A}$.

Let $\mathsf{q}_{\mathcal{D}}$ be the maximum number of calls to $\mathcal{D}$ in our security games. It is said that $\mathcal{D}$ is legitimate if, for all $j \in \{1, \ldots, \mathsf{q}_{\mathcal{D}}\}$,

$$H_\infty(\mathrm{I}_j | \mathrm{I}_1, \ldots, \mathrm{I}_{j-1}, \mathrm{I}_{j+1}, \ldots, \mathrm{I}_{\mathsf{q}_{\mathcal{D}}}, z_1, \ldots, z_{\mathsf{q}_{\mathcal{D}}}, \gamma_1, \ldots, \gamma_{\mathsf{q}_{\mathcal{D}}}) \geq \gamma_j,$$

where H_∞ is the minimum entropy function as defined in Definition 2.1.4.

We model an adversary using a pair $(\mathcal{A}, \mathcal{D})$, where $\mathcal{A}$ is the actual adversary and $\mathcal{D}$ is a stateful distribution sampler. The adversary $\mathcal{A}$'s goal is to determine a challenge bit b picked during the initialise procedure, which also returns the public parameters.

Like past security models, [29] builds upon the security notions of resilience (res), forward security (fwd), backward security (bwd) and robustness (rob), with the latter being the strongest with the most adversarial freedom. Before defining what each notion entails, we describe the game framework used in Figure 2.4 where, as usual, the challenger begins by running the Initialise procedure, and ends with the Finalise procedure.

```
Procedure: Initialise ()
$\sigma \leftarrow 0$,
seed $\xleftarrow{\$}$ setup
$s_0 \xleftarrow{\$}$ setup
$e \leftarrow n$
$corrupt \leftarrow false$
b $\xleftarrow{\$} \{0,1\}$
return seed

Procedure: Finalise (b*)
if b = b* then
    return 1
else
    return 0

Procedure: get-state ()
$e \leftarrow 0$
corrupt ← true
return $s_i$

Procedure: next-ror
$(s_{i+1}, r_0) \leftarrow$ next$(s_i)$
if corrupt = true then
    $e \leftarrow 0$
    return $r_0$
else
if $r_0 = \perp$ then
    $r_1 \leftarrow \perp$
else
    $r_1 \xleftarrow{\$} \{0,1\}^\ell$
return $r_{\mathsf{b}}$

Procedure: get-next
$(s_{i+1}, r_{i+1}) \leftarrow$ next$(s_i)$
if corrupt = true then
    $e \leftarrow 0$
return $r_{i+1}$

Procedure: set-state ($s^*$)
$e \leftarrow 0$
corrupt ← true
$s_i \leftarrow s^*$

Procedure: $\mathcal{D}-$refresh()
$(\sigma, \mathrm{I}, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$
$s_{i+1} \leftarrow$ refresh$(s_i, \mathrm{I})$
$e \leftarrow e + \gamma$
if $e \geq \gamma^*$ then
    $corrupt \leftarrow false$
return $(\gamma, z)$
```

Figure 2.4: The security procedures of [29]. The inclusion of seed has been omitted from several algorithms.

Definition 2.3.2 (Robustness)**.** A PRNG with input is called $(t, \mathsf{q}_{\mathcal{D}}, \mathsf{q}_{\mathsf{R}}, \mathsf{q}_{\mathsf{S}}, \gamma^*, \epsilon_{\mathsf{rob}})$-robust (rob) if, for any attacker $\mathcal{A}$ running in time at most t, making at most $\mathsf{q}_{\mathcal{D}}$ calls to $\mathcal{D}-$refresh, making at most q_{R} calls to next/next-ror, q_{S} calls to get-state/set-state, and any legitimate distribution sampler $\mathcal{D}$, the advantage in the game specified in Figure 2.4 is at most ϵ. The value γ^* is the minimum entropy required to reset the "corrupt" flag back to "false". As usual, the challenger first executes the Initialise algorithm and the adversary is given access to next-ror, get-next, set-state and refresh oracles, described in Figure 2.4. Once the adversary has asked all her queries, she outputs her guess, passes it to the challenger and the challenger runs Finalise with the adversary's guess as input.

Further, we define three more games which are restrictions of the robustness game:

Definition 2.3.3. Resilience (res) is the restricted game where $\mathsf{q}_{\mathsf{S}} = 0$.

Definition 2.3.4. Forward-secure (fwd) is the restricted game where $\mathcal{A}$ makes no calls to set-state and a single call to $\mathsf{get\text{-}state}$, which must be the very *last* oracle call that $\mathcal{A}$ makes.

Definition 2.3.5. Backward-secure (bwd) is the restricted game where $\mathcal{A}$ makes no calls to $\mathsf{get\text{-}state}$ and a single call to $\mathsf{set\text{-}state}$ which is the very *first* call $\mathcal{A}$ makes.

Definition 2.3.6. Let G be a PRNG. For $x \in \{\mathsf{fwd}, \mathsf{bwd}, \mathsf{res}, \mathsf{rob}\}$ as defined in Definitions 2.3.2 to 2.3.5, let $\mathsf{Adv}^{\mathsf{x}}_{\mathsf{G},\mathcal{D}}(\mathcal{A})$ be defined as

$$\mathsf{Adv}^{\mathsf{x}}_{\mathsf{G},\mathcal{D}}(\mathcal{A}) := 2\Pr\left[x^{\mathcal{A}} \implies 1\right] - 1.$$

```
Recover(G, A, D)
$(\mathsf{setup}, \mathsf{refresh}, \mathsf{next}) \leftarrow \mathsf{G}$
$\mathsf{seed} \xleftarrow{\$} \mathsf{setup}; \mathsf{b} \xleftarrow{\$} \{0,1\}$
$\sigma_0 \leftarrow 0; \mu \leftarrow 0$
for $k = 1, \dots, \mathsf{q}_{\mathcal{D}}$ do
  $(\sigma_k, \mathrm{I}_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$
$(s_0, d, \sigma') \xleftarrow{\$} \mathcal{A}^{\mathsf{get\text{-}refresh}}(\mathsf{seed},$
$\gamma_1, \dots, \gamma_{\mathsf{q}_{\mathcal{D}}}, z_1, \dots, z_{\mathsf{q}_{\mathcal{D}}})$
if $\mu + d > \mathsf{q}_{\mathcal{D}}$ or $\sum_{j=\mu+1}^{\mu+d} \gamma_j < \gamma^*$ then
  return 0
for $j = 1, \dots, d$ do
  $s_j \leftarrow \mathsf{refresh}(\mathsf{seed}, s_{j-1}, \mathrm{I}_{\mu+j})$
$(s_0^*, r_0^*) \leftarrow \mathsf{next}(\mathsf{seed}, s_d)$
$s_1 \xleftarrow{\$} \{0,1\}^\ell$
if $r_0^* = \perp$ then
  $r_1^* \leftarrow \perp$
else
  $r_1^* \xleftarrow{\$} \{0,1\}^\ell$
$\mathsf{b}^* \xleftarrow{\$} \mathcal{A}(\sigma', s_{\mathsf{b}}^*, r_{\mathsf{b}}^*, \mathrm{I}_{\mu+d+1}, \dots, \mathrm{I}_{\mathsf{q}_{\mathcal{D}}})$
if $\mathsf{b}^* = \mathsf{b}$ then
  return 1
else
  return 0
```

```
Preserve(G, A)
$(\mathsf{setup}, \mathsf{refresh}, \mathsf{next}) \leftarrow \mathsf{G}$
$\mathsf{seed} \xleftarrow{\$} \mathsf{setup}; s_0 \xleftarrow{\$} \{0,1\}^n; \mathsf{b} \xleftarrow{\$} \{0,1\}$
for $j = 1, \dots, d$ do
  $s_j \leftarrow \mathsf{refresh}(\mathsf{seed}, s_{j-1}, \mathrm{I}_j)$
$(s_0^*, r_0^*) \leftarrow \mathsf{next}(\mathsf{seed}, s_d)$
$s_1 \xleftarrow{\$} \{0,1\}^\ell$
if $r_0^* = \perp$ then
  $r_1^* \leftarrow \perp$
else
  $r_1^* \xleftarrow{\$} \{0,1\}^\ell$
$\mathsf{b}^* \xleftarrow{\$} \mathcal{A}(\sigma', s_{\mathsf{b}}^*, r_{\mathsf{b}}^*)$
if $\mathsf{b}^* = \mathsf{b}$ then
  return 1
else
  return 0

get-refresh ()
$\mu \leftarrow \mu + 1$
return $\mathrm{I}_\mu$
```

Figure 2.5: Preserving and recovering security games for G outputting ℓ-bits.

The main result of [29] is Theorem 1, the composition theorem that proves that if a PWI satisfying two simpler notions of security called preserving and recovering security, then the PWI is robust. We present the associated security games for preserving and recovering security in Figure 2.5. Informally, preserving security states that if the state of the PWI starts uncompromised, and is refreshed using compromised input, then the next output and resulting state are still indistinguishable from random. Similarly, recovering security implies that if a PWI is compromised, inserting enough random entropy to refresh the internal state will ensure that the next output and state will be indistinguishable from random.

Definition 2.3.7 (Preserving Security)**.** A PWI is said to have $(t, \epsilon_{\mathsf{p}})$-preserving security, if, for any adversary $\mathcal{A}$ running in time the preserving advantage defined by

$$\mathsf{Adv}_{\mathsf{G}}^{\mathsf{pres}}(\mathcal{A}) = 2\Pr\left[\mathsf{Preserve}(\mathsf{G}, \mathcal{A}) = 1\right] - 1,$$

satisfies $\mathsf{Adv}_{\mathsf{G}}^{\mathsf{pres}}(\mathcal{A}) \leq \epsilon_{\mathsf{p}}$, for $\mathsf{Preserve}$ as in Figure 2.5.

Definition 2.3.8 (Recovering Security)**.** A PWI is said to have $(t, \mathsf{q}_{\mathcal{D}}, \gamma^*, \epsilon_{\mathsf{r}})$-recovering security, if, for any adversary $\mathcal{A}$ and legitimate sampler $\mathcal{D}$, both running in time t, the recovering advantage defined by

$$\mathsf{Adv}_{\mathsf{G}}^{\mathsf{rec}}(\mathcal{A}) = 2\Pr\left[\mathsf{Recover}(\mathsf{G}, \mathcal{A}, \mathcal{D}) = 1\right] - 1,$$

satisfies $\mathsf{Adv}_{\mathsf{G}}^{\mathsf{rec}}(\mathcal{A}) \leq \epsilon_{\mathsf{r}}$ for $\mathsf{Recover}$ as in Figure 2.5.

Theorem 2.3.9 (Composition theorem of [29])**.**
If a PWI has both $(t, \mathsf{q}_{\mathcal{D}}, \gamma^*, \epsilon_{\mathsf{r}})$-recovering security and $(t, \epsilon_{\mathsf{p}})$-preserving security, then it is $((t', \mathsf{q}_{\mathcal{D}}, \mathsf{q}_{\mathsf{R}}, \mathsf{q}_{\mathsf{S}}), \gamma^*, \mathsf{q}_{\mathsf{R}}(\epsilon_{\mathsf{r}} + \epsilon_{\mathsf{p}}))$-robust where $t' \approx t$.

Theorem 2.3.10 (Composition theorem of [33])**.**
Let $\mathsf{G}[\pi]$ be a PWI that issues ${\mathsf{q}_\pi}^{\mathsf{ref}}$ (resp. ${\mathsf{q}_\pi}^{\mathsf{nxt}}$) π queries in each invocation of $\mathsf{refresh}$ (resp. next); and let $\bar{q}_\pi := \mathsf{q}_\pi + Q(\mathsf{q}_{\mathcal{D}})$. For every $(\mathsf{q}_\pi, \mathsf{q}_{\mathcal{D}}, \mathsf{q}_{\mathsf{R}}, \mathsf{q}_{\mathsf{S}})$-adversary $\mathcal{A}$ against robustness and for every Q-distribution sampler $\mathcal{D}$, there exists a family of $(\mathsf{q}_\pi + \mathsf{q}_{\mathsf{R}} \cdot {\mathsf{q}_\pi}^{\mathsf{nxt}} + \mathsf{q}_{\mathcal{D}} \cdot {\mathsf{q}_\pi}^{\mathsf{ref}})$-adversaries $\mathcal{A}_1^{(i)}$ against recovering security and a family of $(\bar{q}_\pi + \mathsf{q}_{\mathsf{R}} \cdot {\mathsf{q}_\pi}^{\mathsf{nxt}} + \mathsf{q}_{\mathcal{D}} \cdot {\mathsf{q}_\pi}^{\mathsf{ref}})$-adversaries $\mathcal{A}_2^{(i)}$ against preserving security (for $i \in \{1, \ldots, \mathsf{q}_{\mathsf{R}}\}$) such that

$$\mathsf{Adv}_{\mathsf{G}}^{\gamma^*-\mathsf{rob}}(\mathcal{A}, \mathcal{D}) \leq \left(\mathsf{Adv}_{\mathsf{G}}^{(\gamma^*, \mathsf{q}_{\mathcal{D}})-\mathsf{rec}}(\mathcal{A}_1^{(i)}, \mathcal{D}) + \mathsf{Adv}_{\mathsf{G}}^{-\mathsf{pres}}(\mathcal{A}_2^{(i)})\right).$$

2.3.2 Other Models

The later work [30] introduced the idea of a scheduler, inspired by the design of the Fortuna PRNG [31] which aimed at a design to improve the recovery time of a corrupt PRNG by allowing premature get-next queries. We will not be considering a scheduler in this thesis since a design can easily be extended to utilise a scheduler; however, the idea of a scheduler is an interesting prospect in terms of possibly replacing the need for seed, by essentially blinding the entropy inputs in a way the adversary cannot predict, mirroring the purpose of the seed. The scheduler may also help simplify some proofs and allow for weaker refresh functions.

This extension defines several relaxed security notions that modify the security games by removing the conservative reducing of the state entropy to 0 on a premature next query, along with adding counters to aid in determining when a PRNG has recovered from compromise.

Informally a scheduler is a stateful deterministic algorithm taking a scheduler key that partitions entropy inputs into several "pools" that are in themselves PRNGs, in the hope that by refreshing pools in some schedule, one pool will eventually reach enough entropy that refreshing the state of a central simpler PRG with this entropy pool is enough to reset the corrupt flag. The scheduler determines which pool will receive the next entropy input and the next pool that will be emptied when entropy is requested.

Theorem 2 of [30] states that given a secure scheduler, entropy pools G_i, each a robust PRNG, and given a secure PRG that will act as the central component of the PRNG yields a premature-next robust PRNG, meaning it remains secure when an adversary prematurely requests output. In the [29] paper, this kind of adversarial call would result in the entropy estimation of the generator state being reduced to 0.

There are other security models that often target a particular aspect of PRNG security in practise, such as side channel leakage in [59], an extension to [29] in [1], the paper by Terashima and Shrimpton [55] which we will discuss in further detail in Chapter 4, and many others.

2.4 Hash Functions

Hash functions are often used in the construction of PRNGs and universal hash functions will prove very useful in later security proofs.

2.4.1 Preliminaries

Definition 2.4.1. A hash function with output length ℓ is a pair of probabilistic-time algorithms $(\mathsf{Gen}, \mathsf{H})$ satisfying the following:

- Gen is a probabilistic algorithm which takes as input a security parameter 1^λ and outputs a key k. We assume that 1^λ is implicit in k to avoid additional notation;
- H takes as input a key k and a string $x \in \{0,1\}^*$ and outputs a string $\mathsf{H}^k \in \{0,1\}^{\ell(\lambda)}$.

If H^k is only defined for inputs $x \in \{0,1\}^{\ell'(\lambda)}$ and $\ell'(\lambda) > \ell(\lambda)$, then we say that $(\mathsf{Gen}, \mathsf{H})$ is a fixed-length hash function for inputs of length ℓ'. Since $\ell' > \ell$ we also call H a compression function.

2.4.2 Universal Hash Functions

A very useful class of hash function is the family of universal hash functions. These hash functions can be efficiently constructed, see [45, 36]. This makes them a very useful primitive and will be a key component in our security analysis in Section 5.4.

Definition 2.4.2 (Universal Hash Function)**.** Let $H = \{\mathsf{H}_k : \{0,1\}^n \to \{0,1\}^m\}_{k \in \{0,1\}^d}$ be a hash function family, then H is ρ-universal if for any inputs $x \neq x' \in \{0,1\}^n$, we have $\Pr_{k \xleftarrow{\$} \{0,1\}^d}(\mathsf{H}_k(x) = \mathsf{H}_k(x')) \leq \rho$.

We also include two stronger sub-classes of universal hash functions in Definitions 2.4.3 and 2.4.4 from [22].

Definition 2.4.3 (XOR Universal Hash Function)**.** Let $H = \{\mathsf{H}_k : \{0,1\}^n \to \{0,1\}^m\}_{k\in\{0,1\}^d}$ be a universal hash function family, and m be a power of two; then H is XOR universal if $\forall x, y \in \{0,1\}^n, x \neq y$, the value $\mathsf{H}(x) \oplus \mathsf{H}(y) \mod m$ is uniformly distributed in $\{0,1\}^m$.

Definition 2.4.4 (Strong Universal Hash Function)**.** Let $H = \{\mathsf{H}_k : \{0,1\}^n \to \{0,1\}^m\}_{k\in\{0,1\}^d}$ be a universal hash function family, then H is a strongly universal family if $\forall x, y \in \{0,1\}^n, x \neq y$, we have the probability that x, y hash to any pair of hash values h_1, h_2 is

$$\Pr\left[\mathsf{H}_x = h_1 \wedge \mathsf{H}_y = h_2\right] = \frac{1}{2^{2m}},$$

or in other words, is perfectly random.

2.4.3 Leftover Hash Lemma

A very useful tool is the leftover-hash lemma, presented in various literature, such as [29]:

Lemma 2.4.5 (Leftover-Hash Lemma)**.**
Let H be a ρ-universal hash function, where $\rho = (1+\alpha)2^{-m}$ for some $\alpha > 0$. Then for any $k > 0$, it is also a (k,ϵ)-extractor for $\epsilon = \frac{1}{2}\sqrt{2^{m-k} + \alpha}$.

Where an extractor is as defined in Definition 2.1.14. A proof of the lemma can be found in [54, Theorem 8.37].

2.5 Sponge Functions

2.5.1 The Two Phases of the Sponge

The sponge construction was introduced in [16] and given a formal security analysis in [17]. The sponge design has benefited from a large amount of analysis due to the success of Keccak in the SHA3 competition in 2012. The sponge construction is stateful and has a $b = (r + c)$-bit state (called the width) split into the inner state of c-bits (the capacity) and outer state of r-bits (the rate).The sponge construction makes use of a keyless permutation π that maps b-bits to b-bits, which is modelled as an ideal permutation in the security analysis.

The normal running of the sponge is split into two phases: the absorbing phase where it incorporates new input into the state and the squeezing phase, where the sponge outputs a specified number of bits.

Algorithm 1 The absorbing function $\mathsf{absorb}[\pi, r](\mathrm{I})$

```
Require: $r < b$
  Interface: $s \leftarrow \mathsf{absorb}(\mathrm{I})$ with $\mathrm{I} \in \mathbb{Z}_2^*$ and $s \in \mathbb{Z}_2^b$.
  $P \leftarrow \mathrm{I} \| \mathrm{pad}[r](|M|)$
  $s \leftarrow 0^b$
  for $i = 0$ to $|P|_r$-1 do
      $s \leftarrow s \oplus (P_i \| 0^{b-r})$
      $s \leftarrow \pi(s)$
  end for
  return $s$
```

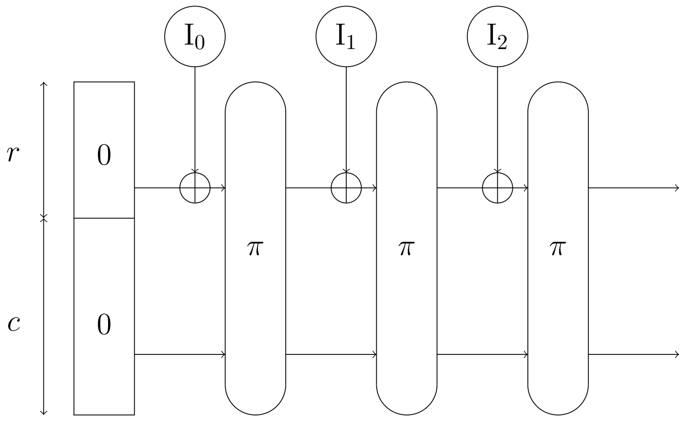


Figure 2.6: The sponge construction in the $\mathsf{absorbing}$ phase.

When initialising the sponge the first state is set to zero. The input message is padded using a valid padding scheme and cut into blocks of r-bits which are then absorbed in the $\mathsf{absorbing}$ phase of the sponge as described in Algorithm 1 and described pictorially in Figure 2.6.

Next, once this initial input has been absorbed the sponge switches to the $\mathsf{squeezing}$ phase as described in Algorithm 2 and shown in Figure 2.7. This algorithm is the dual to the $\mathsf{absorbing}$ function. For a given state s and requested number of bits ℓ it outputs a string truncated to ℓ from the sponge function acting on state s at the beginning of the $\mathsf{squeezing}$ phase. Once the $\mathsf{squeezing}$ phase is completed, the sponge function finishes.

Algorithm 2 The squeezing function $\mathsf{squeeze}[\pi, r](s, \ell)$

```
Require: $r < b$
  Interface: $Z \leftarrow \mathsf{squeeze}(s, \ell)$ with $s \in \mathbb{Z}_2^b$, integer $\ell > 0$ and $Z \in \mathbb{Z}_2^\ell$
  $Z \leftarrow \lfloor s \rfloor_r$
  while $r|Z|_r < \ell$ do
    $s \leftarrow \pi(s)$
    $Z \leftarrow Z \| \lfloor s \rfloor_r$
  end while
  Return $\lfloor Z \rfloor_\ell$
```

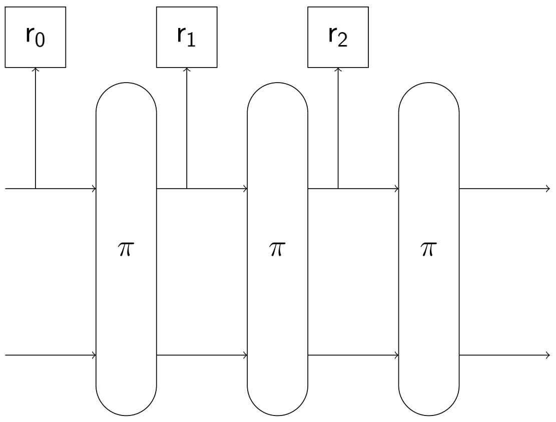


Figure 2.7: The sponge construction in the squeezing phase.

Algorithm 3 The sponge construction $\mathsf{sponge}[\pi, \mathsf{pad}, r](M, \ell)$

```
Require: $r < b$
  Interface: $Z \leftarrow \mathsf{sponge}(M, \ell)$ with $M \in \mathbb{Z}_2^*$, integer $\ell > 0$ and $Z \in \mathbb{Z}_2^\ell$
  $P \leftarrow M \| \mathsf{pad}[r](|M|)$
  $s \leftarrow 0^b$
  for $i = 0$ to $|P|_r$ do
      $s \leftarrow s \oplus (P_i \| 0^{b-r})$
      $s \leftarrow \pi(s)$
  end for
  $Z \leftarrow \lfloor s \rfloor_r$
  while $r|Z|_r < \ell$ do
      $s \leftarrow \pi(s)$
      $Z \leftarrow Z \| \lfloor s \rfloor_r$
  end while
  return $\lfloor Z \rfloor_\ell$
```

2.5.2 The Sponge and Duplex Algorithms

In full, the sponge construction is given in Algorithm 3 and shown in Figure 2.8, it has precisely one absorbing phase and one squeezing phase.

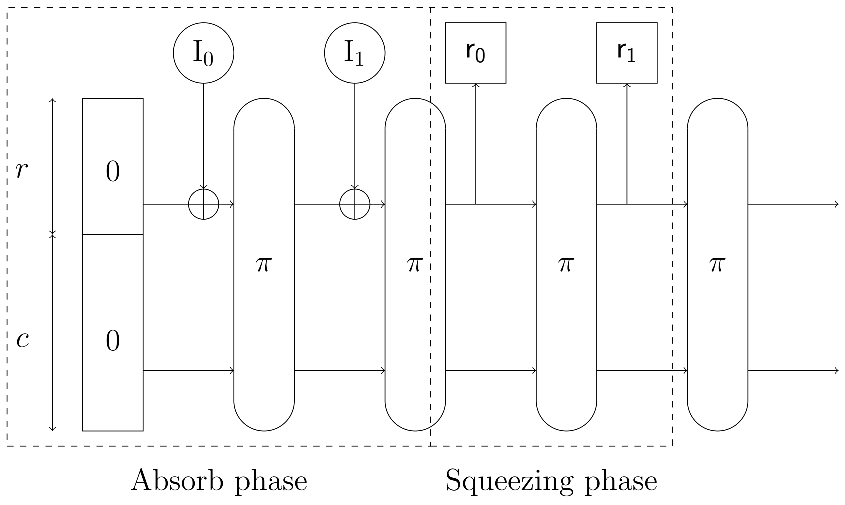


Figure 2.8: The sponge construction in both phases.

Algorithm 4 The duplex construction $\mathsf{duplex}[\pi, \mathsf{pad}, r]$

```
Require: $r < b$
Require: $\rho_{\max}(\mathsf{pad}, r) > 0$
  Interface: $\mathcal{D}.\mathsf{Initialise}\ ()$
  $s \leftarrow 0^b$
  Interface: $Z \leftarrow \mathcal{D}.\mathsf{duplex}(\sigma, \ell)$ with $\ell \leq r, \sigma \in \cup_{n=0}^{\rho_{\max}}(\mathsf{pad}, r)\mathbb{Z}_2^b$, and $Z \in \mathbb{Z}_2^\ell$
  $P \leftarrow \sigma \| \mathsf{pad}[r](|\sigma|)$
  $s \leftarrow s \oplus (P \| 0^{b-r})$
  $s \leftarrow \pi(s)$
  return $\lfloor s \rfloor_\ell$
```

The duplex construction, given in [16] differs slightly in that it has multiple absorbing and squeezing phases and will be useful when defining the PRNG variant of the sponge construction. The duplex construction or duplex mode is given in Algorithm 4 and shown in Figure 2.9. Since the duplex construction has multiple absorbing and squeezing phases, it begins by initialising the state to 0 followed by all interactions taking place through duplex calls. This allows the construction to continue indefinitely.

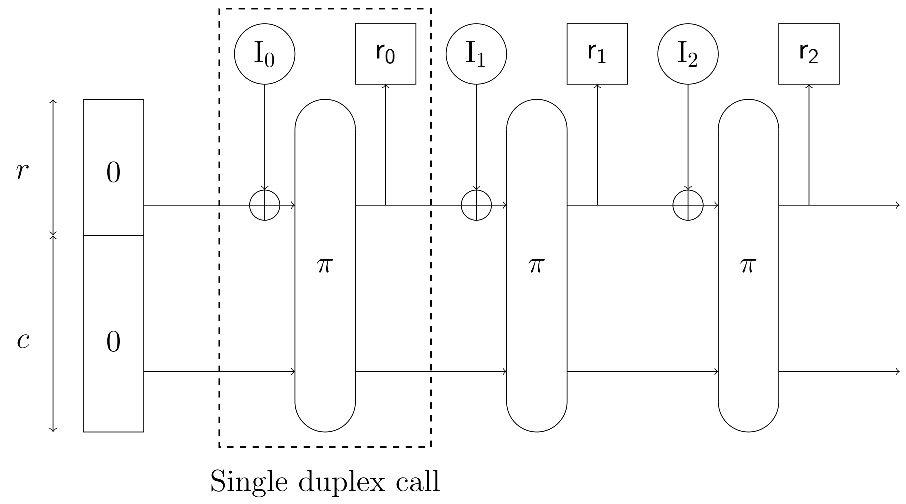


Figure 2.9: The sponge construction in duplex mode.

2.5.3 The Sponge PRNG: sponge.prng

The authors of [18] use the sponge construction, specifically the duplex mode, to build a PRNG, which we present below in Algorithm 5. We call this PRNG the sponge.prng and critique it in Chapter 3. The sponge.prng relies heavily on the duplex construction, with the addition of an optional subroutine called p.forget that zeroes the outer state by XORing the current outer state. The p.forget subroutine can be optionally used in an implementation after each fetch request is made to give some measure of forward security.

Algorithm 5 The sponge PRNG construction sponge.prng$[\pi, \mathsf{pad}, r, \rho]$

```
Require: $\rho \leq \rho_{\max}(\mathsf{pad}, r) - 1$
Require: $D = \mathsf{duplex}[\pi, \mathsf{pad}, r]$

  Interface: $P$.Initialise ()
  $D$.Initialise ()
  $B_{in}$ = empty string
  $B_{out}$ = empty string

  Interface: $P.\mathsf{feed}(\sigma)$ with $\sigma \in \mathbb{Z}_2^+$
  $M = B_{in} \| \sigma$
  for $i = 0$ to $|M|_\rho - 2$ do
      $D.\mathsf{duplex}(M_i, 0)$
  end for
  $B_{in} = M_{|M|_\rho} - 1$
  $B_{out}$ = empty string

  Interface: $Z = P.\mathsf{fetch}(\ell)$ with integer $\ell \geq 0$ and $Z \in \mathbb{Z}_2^\ell$
  while $|B_{out}| < \ell$ do
      $B_{out} = B_{out} \| D.\mathsf{duplex}(B_{in}, \rho)$
      $B_{in}$ = empty string
  end while
  $Z = \lfloor B_{out} \rfloor_\ell$
  $B_{out}$ = last$(|B_{out}| - \ell)$ bits of $B_{out}$
  return $Z$

  Interface: p.forget ()
  $Z = D.\mathsf{duplex}(B_{in}, \rho)$
  $B_{in}$ = empty string
  for $i = 1$ to $\lfloor c/\rho \rfloor$ do
      $Z = D.\mathsf{duplex}(Z, \rho)$
  end for
  $B_{out}$ = empty string
```

2.6 Generalising the Sponge to the Parazoa Family

The sponge construction is generalised by Andreeva, Mennink and Preneel in [3]. Similar to the sponge, the parazoa function is split into two sections; compression/absorbing phase where messages or input is padded and absorbed into an n-bit state, and the extraction/squeezing phase when output is generated.

The parazoa utilises two main functions f and g; the compression phase refers directly to the function f, while the extraction phase refers to the g function. Each function is of a specific form and is built up of two inner functions, which are referred to as f_1, f_2, g_1 and g_2. Both functions are based on a permutation π. The parazoa also requires a padding scheme and finalise function. The padding scheme takes a message input I and pads it into several input blocks of p-bits. The finalise function takes the ℓ-bit outputs and combines them into the ℓ_{total}-bit digest.

Definition 2.6.1. A function $f : \mathbb{Z}_2^a \longrightarrow \mathbb{Z}_2^b$ for $a \geq b$ is called *balanced* if $\forall y \in \mathbb{Z}_2^b$, y has exactly 2^{a-b} pre-images under f.

Definition 2.6.2. For $x \in \mathbb{Z}_2^n$, define the capacity set

$$C(x) := \{s \in \mathbb{Z}_2^n \mid \exists \mathrm{I} \in \mathbb{Z}_2^p \text{ s.t. } f_1(s, \mathrm{I}) = x\}.$$

2.6.1 The f Function

Definition 2.6.3. The f function takes on input the current value of the n-bit state s_{i-1} and the message block I_i:

$$f : \mathbb{Z}_2^n \times \mathbb{Z}_2^p \longrightarrow \mathbb{Z}_2^n.$$

The first of the sub functions f_1 absorbs the message into the state and then permutes the state with π, before the second sub function f_2 transforms the state while

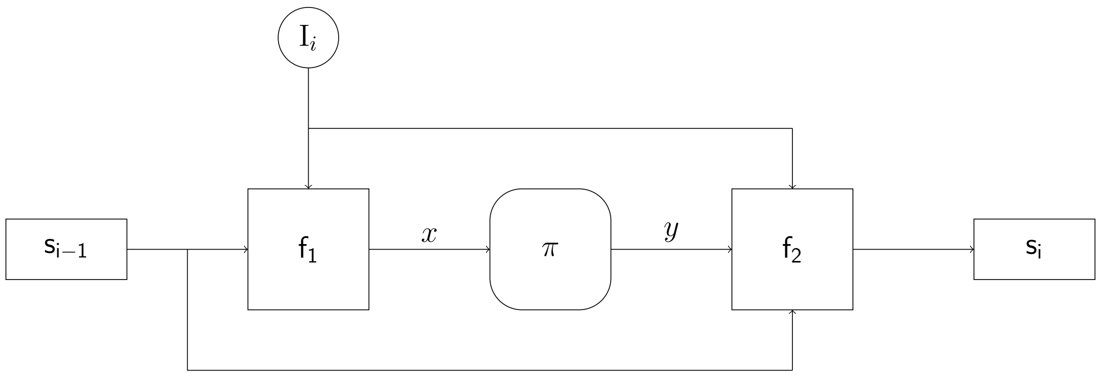


Figure 2.10: The f function.

possibly combining it with a feed-forward. More formally:

$$f_1 : \mathbb{Z}_2^n \times \mathbb{Z}_2^p \longrightarrow \mathbb{Z}_2^n$$
$$f_2 : \mathbb{Z}_2^n \times \mathbb{Z}_2^n \times \mathbb{Z}_2^p \longrightarrow \mathbb{Z}_2^n$$
$$s_i \leftarrow f(s_{i-1}, \mathrm{I}_i)$$
$$x \leftarrow f_1(s_{i-1}, \mathrm{I}_i)$$
$$y \leftarrow \pi(x)$$
$$s_i \leftarrow f_2(y, s_{i-1}, \mathrm{I}_i).$$

Figure 2.10 gives a more intuitive picture of how f operates. For further clarity, in the case of the sponge construction, $x = f_1(s_{i-1}, \mathrm{I}_i) = s_{i-1} \oplus (\mathrm{I}_i \| 0^c)$ and $s_{i+1} = f_2(y, \mathrm{I}_i, s_{i-1}) = y$, the identity map. It is also required that the f_1 and f_2 functions satisfy the following:

- The function f_1 must satisfy the following properties:
  1. $\forall x \in \mathbb{Z}_2^n, \forall s \in C(x), \exists! \mathrm{I}$ s.t. $f_1(s, \mathrm{I}) = x$ (uniqueness).
  2. $\forall x, x' \in \mathbb{Z}_2^n$, if $C(x) \cap C(x') \neq \varnothing$ then $C(x) = C(x')$.
- The function f_2 must be a bijection on the state.

2.6.2 The g Function

Definition 2.6.4. The g function takes the current value of the state s_{k+i-1} as input:

$$g : \mathbb{Z}_2^n \to \mathbb{Z}_2^n \times \mathbb{Z}_2^\ell.$$

The first of the sub functions g_1 outputs a block and permutes the state with π. A second sub function g_2 transforms this intermediate state (or in fact for simpler proofs may do nothing) using the previous state as additional input. More formally:

$$\begin{aligned}
&g_1 : \mathbb{Z}_2^n \longrightarrow \mathbb{Z}_2^n \times \mathbb{Z}_2^\ell \\
&g_2 : \mathbb{Z}_2^n \times \mathbb{Z}_2^n \longrightarrow \mathbb{Z}_2^n \\
&(s_{k+i}, r_{k+i-1}) \leftarrow g(s_{k+i-1}) \\
&(s^*_{k+i-1}, r_{k+i-1}) \leftarrow g_1(s_{k+i-1}) \\
&s^{**}_{k+i-1} \leftarrow \pi(s^*_{k+i-1}) \\
&s_{k+i} \leftarrow g_2(s^{**}_{k+i-1}, s_{k+i-1}).
\end{aligned}$$

A more intuitive picture of how g operates is given in Figure 2.11, though initially g_2 is the identity, but can be generalised as shown in Figure 2.12. For clarity, the sponge function can be thought of as having g_1 output the outer state while leaving the state itself unchanged, along with g_2 as the identity map on y. It is also required that the g_1 and g_2 functions satisfy the following:

- The function g_1 is a bijection on the state, and when restricted to viewing the output r_i, g_1 is balanced.
- The function g_2 must also be a bijection on the state.

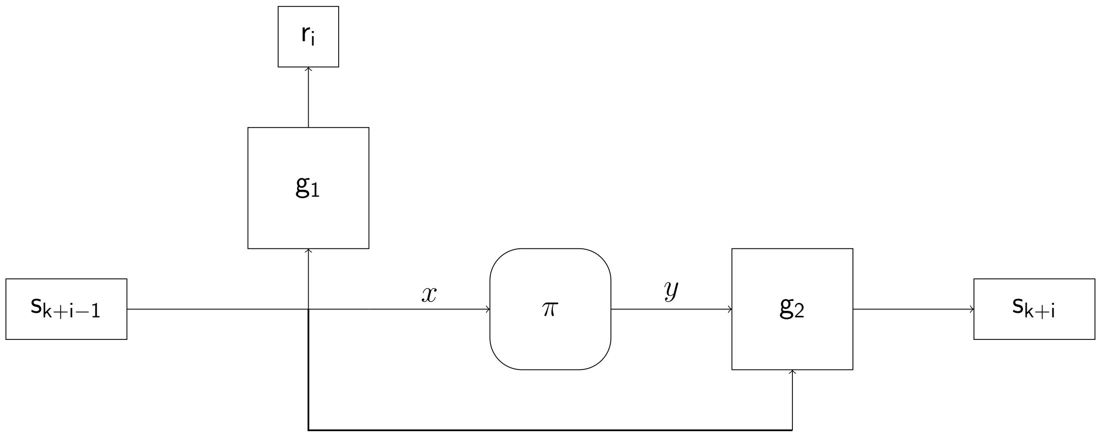


Figure 2.11: The g function.

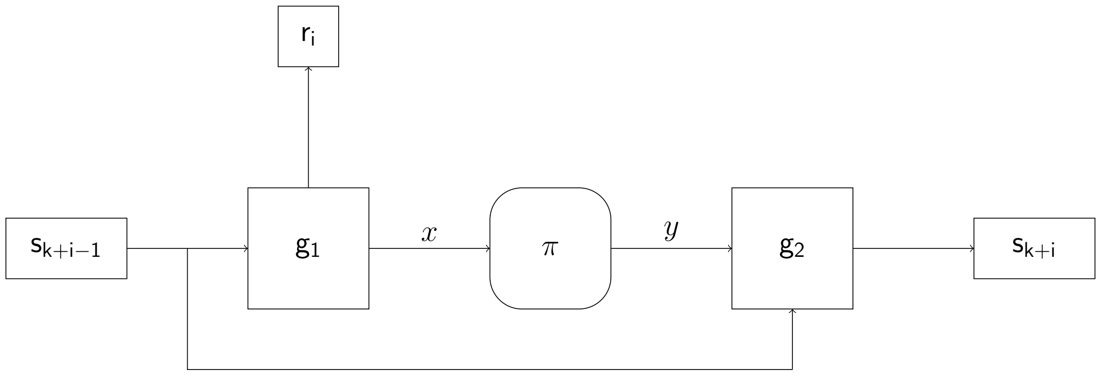


Figure 2.12: The generalised g function.

The f and g functions can be further generalised by using two different permutations π_1 and π_2 respectively.

Definition 2.6.5. The padding function pad is an injective mapping that takes the input messages of arbitrary length and transforms them into blocks of length p. It is required that pad satisfies the following property:

- Either, $\ell_b = 1$, or
- the last block of a message I_k satisfies $\forall s \in \mathbb{Z}_2^n, \forall (s', \mathrm{I}') \in \mathbb{Z}_2^n \times \mathbb{Z}_2^p$,

$$f_1(s, M_k) \neq s \text{ and } f_1(f_2(s, s', \mathrm{I}'), \mathrm{I}_k) \neq s.$$

Definition 2.6.6. The finalise function fin combines the ℓ_b bit strings obtained from squeezing the parazoa, into the final output. The fin function must be balanced as per Definition 2.6.1.

The sponge function uses a finalise function that concatenates the outputs and truncates this concatenation to the correct length, which trivially meets the balanced requirement.

2.6.3 Formal Definition of a Parazoa Function

Definition 2.6.7. A parazoa, denoted $\mathcal{P}$, is defined as a tuple $(\ell_b, p, \ell_{\mathsf{total}}, \ell, n)$ along with a padding scheme pad as described in Definition 2.6.5, finalise function fin as described in Definition 2.6.6, together with functions f and g as described in Definitions 2.6.3 and 2.6.4. The tuple reads as follows:

- ℓ_b is the number of output data blocks required to output each ℓ_{total} request.
- p is the size of each input block to the f_1 function, denoted I_i.
- ℓ_{total} is the size of the output of the fin function.
- ℓ is the size of each output block of the g_1 function, denoted r_i.
- n is the size of the internal state, denoted s_i.

Along with the individual component requirements given in each definition, it is required that $p, \ell \leq n$ and $\ell\ell_b \geq \ell_{\mathsf{total}}$.

We note that the number of output data blocks ℓ_b may be a larger number due to complex finalise functions fin that may further compress the output. A technical variable useful for describing the security of a parazoa $\mathcal{P}$ in regards to indifferentiability to a random oracle $\mathcal{RO}$ is the following definition of capacity loss d.

Definition 2.6.8. Consider the set of all pairs $(s, x) \in \mathbb{Z}_2^n \times \mathbb{Z}_2^n$ such that $f_1(s, \mathrm{I}) = x$ for some $\mathrm{I} \in \mathbb{Z}_2^p$ uniquely determined by (s, x). The *capacity* loss of a parazoa $\mathcal{P}$ denoted $d \geq 0$, is defined as the minimum of

$$\text{For } x \in \mathbb{Z}_2^n, r \in \mathbb{Z}_2^r, \text{ there are at most } 2^d \text{ pairs } (s,x) \text{ such that } s \in \{g_1^{-1}(r)\};$$
$$\text{For } s \in \mathbb{Z}_2^n, r \in \mathbb{Z}_2^r, \text{ there are at most } 2^d \text{ pairs } (s,x) \text{ such that } x \in \{g_1^{-1}(r)\}.$$

In particular, restricting to the f_1 and f_2 used by the sponge function, along with g_1 output function of truncating the outer state, yields $d = 0$. The authors of [3] extend the indifferentiability of the sponge from a random oracle to the parazoa in the following theorem.

Theorem 2.6.9 (Theorem 1 of [3]).
Let π be a random n-bit permutation, and let $\mathcal{RO}$ be a random oracle. Let $\mathcal{P}$ be a parazoa function parameterised by $(\ell_b, p, \ell_{\mathsf{total}}, \ell, n)$. Let D be a distinguisher that makes at most q_1 left queries of maximal length $(K-1)p$-bits, where $K \geq 1$, and q_2 right queries, and runs in time t. Then for a simulator sim making at most $\mathsf{q_S} \leq q_2$ queries to $\mathcal{RO}$ and running in time $O(q_2^2)$, we have:

$$\mathsf{Adv}^{\mathsf{pro}}_{\mathcal{P},\mathsf{sim}}(\mathsf{D}) = O\left(\frac{((K+\ell_b)q_1 + q_2)^2}{2^{n-\ell-d}}\right),$$

where a simulator is formally defined in Definition 2.7.4.

2.7 Proof Techniques

2.7.1 Indifferentiability

The notion of indifferentiability is the more general form of indistinguishability given by Maurer et al in [42], with works such as [25] applying it to relevant areas such as hash functions.

Definition 2.7.1. An $(\mathcal{X}, \mathcal{Y})$-system is a sequence of probability distributions $P_{Y_i|X^iY^{i-1}}$ for $i \in \mathbb{N}$, where $X^i := [X_1, \ldots, X_i]$ and $Y^{i-1} := [Y_1, \ldots, Y_{i-1}]$ and where X_i is called the ith input, and Y_i the ith output, are random variables with range $\mathcal{X}$ and $\mathcal{Y}$ respectively. If each Y_i only depends on the actual input X_i, and possibly some randomness, then the system is called a random function.

Definition 2.7.2. Let $S = (S_k)_{k\in\mathbb{N}}$ and $T = (T_k)_{k\in\mathbb{N}}$ be two $(\mathcal{X}, \mathcal{Y})$-systems, then S and T are computationally indistinguishable if for any computationally efficient distinguisher D, interacting with one of these systems and generating a binary output, the advantage

$$|\Pr[D(S_k) \to 1] - \Pr[D(T_k) \to 1]| \leq \mathsf{negl}(k),$$

for security parameter k.

The following proposition describes how this notion is utilised in security reductions:

Proposition 2.7.3.
For S and T defined above, it is said S and T are indistinguishable if and only if for every cryptosystem $C(T)$ using T as a component, the cryptosystem $C(S)$ obtained from $C(T)$ by replacing the component T with S is at least as secure as $C(T)$.

It's important to note that this proposition only applies when the resources involved have no public interfaces, i.e an adversary has no direct access to these components.

It is often the case that an adversary will have more access than this; she may have access to another interface which the first interface interacts with, such as a permutation a construction utilises. We label the interface with the construction to be the private interface (or interface 1) while the primitive this interface utilises is called the public interface (or interface 2). This motivates the definition of indifferentiability:

Definition 2.7.4. For S and T defined above, it is said that S is indifferentiable from T, if for any distinguisher D with binary output there is a system P such that the advantage

$$|\Pr\left[\mathsf{D}(S_k^1, S_k^2) \to 1\right] - \Pr\left[\mathsf{D}(T_k^1, P(T_k^2)) \to 1\right]| \leq \mathsf{negl}(k),$$

for security parameter k. This is easier to understand using Figure 2.13:

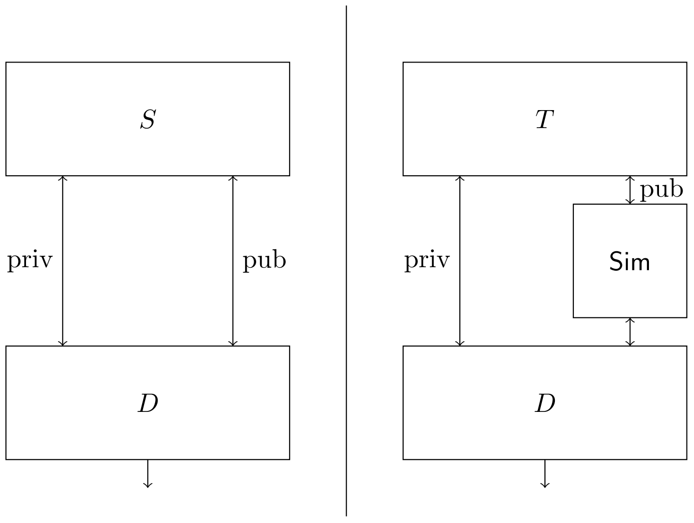


Figure 2.13: The distinguisher D differentiating between S and T is either connected as in the left or right. In the first case D has direct access to both the public and private interfaces of S, while in the latter case the access to the public interface of T is replaced by an intermediate system P (called sim) that can make its own calls to the public interface of T.

An example of differentiability

For clarity we give an example of how two constructions can easily be differentiable due to the interfaces the adversary has access to.

Theorem 2.7.5.
Let f, g be random functions on $\{0,1\}^n$ and let f^2 denote f applied iteratively twice. Then, f^2, f is differentiable from $g, \mathsf{sim}[g]$, or, in other words, there exists no efficient simulator that can fool the proposed distinguisher D.

The idea behind Theorem 2.7.5 can be seen clearer in Figure 2.14.

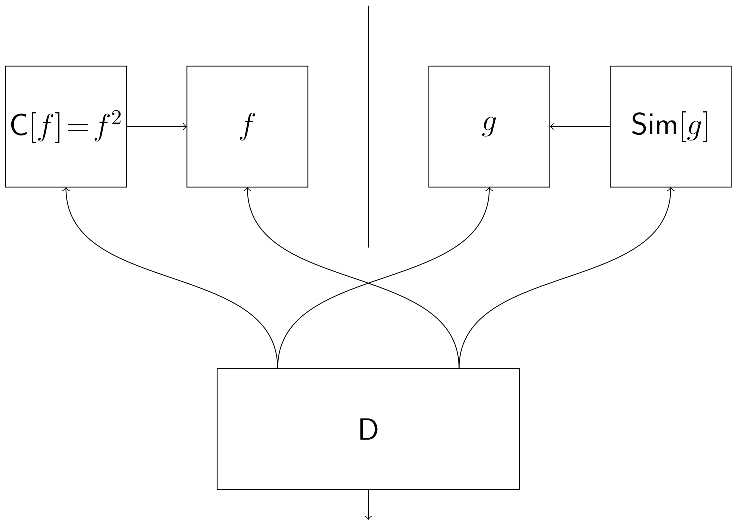


Figure 2.14: The interfaces available to the distinguisher D.

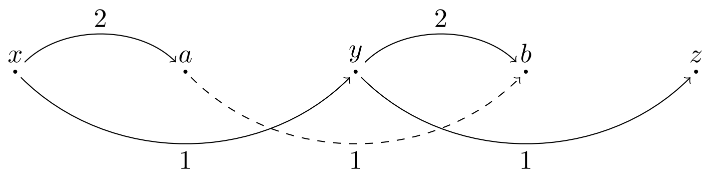


Figure 2.15: Illustrating the method D uses to cause a contradicting output from the simulator.

Proof. The Distinguisher is defined as follows: Query the left interface with some $x \xleftarrow{\$} \{0,1\}^n$, with y the result of this query. D queries the left interface again with y, and receives the response z.

Next, D queries the right interface with y and will receive the response b. If the right interface is the simulator, it can easily deduce that whatever it outputs must be linked to z, by querying the left interface with y. This is so that $f(y) = b$ and $f(b) = z$ to maintain consistency.

The distinguisher now queries the left interface with x, which outputs a value a. The simulator can deduce that x is linked to y, which is linked to b, but with overwhelming probability the output a will not be correct if queried to the left interface. This is due to $g(a) \neq b$ with overwhelming probability $(1/2^n)$. □

Corollary 2.7.6.
This attack does not apply if f, g are random permutations and the construction allows inverse queries, since the simulator can query $(f^2/g)^{-1}(b) := a$ and link this value with x and y.

2.7.2 H-coefficient Technique

This section gives a brief introduction to Patarin's H-coefficient technique with a focus on its use in cryptographic proofs. Influenced by [23] and initially defined in [47], the H-coefficient technique is applied by splitting the "transcripts" of a game into two or more distinct sets; calculating the probability of the real or ideal world outputting transcripts in a particular set yields a close bound for the statistical distance of the real and ideal world.

A high level overview is that of a q-query information theoretic adversary $\mathcal{A}$ which can be assumed to be deterministic, making no redundant queries without loss of generality, interacting with an oracle ω representing either the real world or ideal world. The interaction $\mathcal{A}$ has with this oracle ω is represented in a transcript τ, which includes a list of queries and their answers given by ω.

Let ω be an oracle that serves as the way the adversary $\mathcal{A}$ interacts with the challenger in the chosen world. Let Ω_X refer to the probability space of all real world oracles with the uniform probability distribution, and similarly Ω_Y is the probability space of all ideal world oracles again with the uniform distribution.

Let $\mathcal{T}$ be the set of all transcripts, with $\tau \in \mathcal{T}$ an individual transcript that describes, in full, the interactions and final output between the adversary $\mathcal{A}$ and the oracle she interacts with.

Further, the random variables X and Y are defined over the probability spaces Ω_X and Ω_Y respectively. We write $X(\omega) = \tau$ to refer to running $\mathcal{A}$ on oracle ω for $\omega \in \Omega_X$, which in turn produces the transcript τ. The random variable Y is defined similarly using Ω_Y. Alternatively X and Y are the functions

$$X : \Omega_X \longrightarrow \mathcal{T}, \qquad\qquad Y : \Omega_Y \longrightarrow \mathcal{T}$$
$$\omega \longmapsto \tau, \qquad\qquad \omega \longmapsto \tau.$$

If we fix a distinguisher $\mathcal{A}$, we can say that $\mathcal{A}$'s distinguishing advantage is upper bounded by

$$\mathsf{SD}(X, Y) = \frac{1}{2} \sum_{\tau \in \mathcal{T}} \left| \Pr\left[X = \tau\right] - \Pr\left[Y = \tau\right] \right|.$$

For simplicity we will only consider two sets; good and bad transcripts, which are denoted $\mathcal{T}_{\mathsf{Good}}$ and $\mathcal{T}_{\mathsf{Bad}}$ respectively.

We say that an oracle $\omega \in \Omega_X$ is compatible with a transcript τ, and denote by $\mathsf{comp}_X(\tau)(\tau)$ the set of such oracles. Similarly, we denote by $\mathsf{comp}_Y(\tau)(\tau)$ the oracles $\omega \in \Omega_Y$ compatible with τ. It should be noted that this does not mean running $\mathcal{A}$ with ω will always produce τ, since it may be the case that the particular transcript cannot be output by $\mathcal{A}$, e.g. it contains more queries than $\mathcal{A}$ is allowed to make, i.e. $\Pr\left[X = \tau\right] = \Pr\left[Y = \tau\right] = 0$. The notion of compatible transcripts is often used in the H-coefficient technique by taking advantage of the following:

$$\Pr\left[X = \tau\right] = \frac{|\mathsf{comp}_X(\tau)|}{|\Omega_X|} \text{ and } \Pr\left[Y = \tau\right] = \frac{|\mathsf{comp}_Y(\tau)|}{|\Omega_Y|},$$

if either $\Pr\left[X = \tau\right] > 0$ or $\Pr\left[Y = \tau\right] > 0$, and noting that

$$\Pr\left[X = \tau\right] > 0 \implies \Pr\left[Y = \tau\right] > 0.$$

These quantities actually have some very interesting consequences. We note that the right hand sides are in fact independent of the choice of $\mathcal{A}$, which actually means that two adversaries produce two different transcripts τ_1, τ_2 that contain the same set of queries, possibly in different orders, they do so with the same probability.

An alternative way to calculate these quantities that sometimes may prove simpler is as follows:

$$\frac{|\mathsf{comp}_X(\tau)|}{|\Omega_X|} = \Pr_{\Omega_X}[\omega \in \mathsf{comp}_X(\tau)] \text{ and } \frac{|\mathsf{comp}_Y(\tau)|}{|\Omega_Y|} = \Pr_{\Omega_Y}[\omega \in \mathsf{comp}_Y(\tau)].$$

Defining the split of transcripts is integral to the proof since the H-coefficient technique allows bounding the statistical distance of the random variables X and Y in the following way: suppose $\exists \epsilon \in [0,1]$, such that $\forall \tau \in \mathcal{T}_{\mathsf{Good}}$, with $\Pr[Y = \tau] > 0$,

$$\frac{\Pr[X = \tau]}{\Pr[Y = \tau]} = \frac{\Pr_{\Omega_X}[\omega \in \mathsf{comp}_X(\tau)].}{\Pr_{\Omega_Y}[\omega \in \mathsf{comp}_Y(\tau)]} \geq 1 - \epsilon.$$

Finally, this culminates in the fundamental theorem of the H-coefficient technique as presented in [23]:

Theorem 2.7.7 (Equation 10)**.**
Let $X, Y, \mathcal{T}_{\mathsf{Good}}, \mathcal{T}_{\mathsf{Bad}}, \tau, \epsilon$ *be as above, then,*

$$\mathsf{SD}(X, Y) \leq \epsilon + \Pr[Y \in \mathcal{T}_{\mathsf{Bad}}].$$

CHAPTER 3

A New Sponge-like PRNG with Analysis

Contents

This chapter was published as a paper at SAC2016 and was originally done concurrently to work by Gaži and Tessaro [33] on improving the sponge.prng design which was published first and inspired several changes in our SAC2016 paper before its eventual submission. We compare our design to their updated design. This paper was also done concurrently to work by Andreeva, Daemen, Mennink and van Assche [2] who also decided to apply the H-coefficient technique to the sponge construction to prove security without using a differentiability proof.

We originally began work on modifying and improving the original sponge-based design sponge.prng, as introduced in [18] and as presented in Section 2.5.3 and Section 3.2.1. We designed an improved forward security measure and decided to call the design Reverie. We initially tried to apply the indifferentiability framework to analyse the design, however we discovered this was not applicable, as discussed in Section 3.2.3. We then began modifying the security model and proved security using the H-coefficient technique. Shortly afterwards, [33] was published and we decided to adapt their security model modifications, in part to allow for easier comparison. We altered our proofs to reflect this.

The design of Reverie draws on the Davies-Meyer construction [56, 26] for inspiration for the feed-forward design, matching the simplicity of the sponge design. The design was also influenced by the extensive analysis in [50, 19]. Introducing this measure invalidates the design's status as a sponge which affects the applicability of the generic sponge security guarantee. We could have instead reformulated the sponge security guarantee by proving indifferentiability of the new design with an "ideal" PRNG.

This chapter first dictates the necessary preliminaries and the original sponge.prng design before we present the sponge-like design Reverie. We also provide the updates to the security model [29] from [33] that reflect the increased adversarial access afforded to the adversary in the security games due to the public random permutation that is present in the sponge-based setting.

3.1 Preliminaries

Here we provide a reminder of the necessary definitions and notions before proceeding to the designs. More detail can be found in Sections 2.3 and 2.5.

3.1.1 Updates to Security Notions

Recall the following basic specification and notation of the sponge construction.

- A sponge is stateful, with an n-bit state s_i.
- The state s usually denoted with an identifier such as s_i, to refer to the i-th state, is split into an inner state of c-bits, denoted by $\widehat{s}_i$ and outer state of r-bits and is denoted $\bar{s}_i$.
- We will often write the state as $s_i = (\bar{s}_i \| \widehat{s}_i)$ where $\|$ is the usual concatenation of strings.
- The construction defined in this thesis utilises a public, random permutation π from the set $\mathcal{P}_n$ of all permutations on n-bits.
- We denote by r_{i+1} the output associated with the ith call to the generator using next, which we denote as $\mathsf{next}(s_i) = (s_{i+1}, r_{i+1})$.

We require a slightly modified definition of a distribution sampler which we give below in Definition 3.1.1, followed by an updated notion of what it means for a distribution sampler $\mathcal{D}$ to be legitimate, given in Definition 3.1.2.

Definition 3.1.1 (Originally from [29] but as amended in [33])**.** A Q-distribution sampler is a randomised stateful oracle algorithm $\mathcal{D}$ which operates as follows:

- It takes a state σ_i, with initial state $\sigma_0 = \perp$.
- $\mathcal{D}^\pi(\sigma_i)$ outputs a tuple $(\sigma_i, \mathcal{S}_i, \gamma_i, z_i)$, where
  - σ_i is the new state of $\mathcal{D}^\pi$,
  - $\mathcal{S}_i$ is a source with range $[\mathcal{S}_i] \subseteq \{0,1\}^{\ell_i}$ for some $\ell_i \geq 1$,
  - γ_i is an entropy estimation for $\mathcal{S}_i$ which will be discussed further below,
  - z_i is the leakage and/or auxiliary information about $\mathcal{S}_i$.
- When run $\mathsf{q}_\mathcal{D}$ times, the number of queries to the permutation π made by $\mathcal{D}^\pi$ and $\mathcal{S}_1, \ldots, \mathcal{S}_{\mathsf{q}_\mathcal{D}}$ is at most $Q(\mathsf{q}_\mathcal{D})$.

For simplicity, $(\sigma_i, \mathrm{I}_i, \gamma_i, z_i) \overset{\$}{\leftarrow} \mathcal{D}^\pi(\sigma_{i-1})$ is written as the overall process of running $\mathcal{D}$ and the generated source $\mathcal{S}_i$. Next, we will require restriction to a certain class of distribution samplers to avoid trivial wins for the adversary and to allow for the public random permutation. We start with the following game, as given in [33, Definition 3].

Let $\mathcal{D}$ be a distribution sampler, $\mathcal{A}$ an adversary and fix an $i^* \in [\mathsf{q}_\mathcal{D}]$. Let $Q_\mathcal{D}$ be the set of all input-output pairs of permutation queries made by $\mathcal{D}$ and by all $\mathcal{S}_j$ for $j \in [\mathsf{q}_\mathcal{D}]/\{i^*\}$.

Then $\mathcal{D}$ is said to be a $(\mathsf{q}_\mathcal{D}, \mathsf{q}_\pi)$-legitimate distribution sampler if for every adversary $\mathcal{A}$ making q_π queries and every $i^* \in [\mathsf{q}_\mathcal{D}]$, all possible values of

$$(\mathrm{I}_j)_{j \in [\mathsf{q}_\mathcal{D}]/(i^*)}, (\gamma_1, z_1), \ldots, (\gamma_{\mathsf{q}_\mathcal{D}}, z_{\mathsf{q}_\mathcal{D}}), V_\mathcal{A}, Q_\mathcal{D},$$

potentially output by the game given in Figure 3.1 with positive probability,

$$\Pr\left[\mathrm{I}_{i^*} = x \mid (\mathrm{I}_j)_{j \neq i^*}, (\gamma_1, z_1), \ldots, (\gamma_{\mathsf{q}_\mathcal{D}}, z_{\mathsf{q}_\mathcal{D}}), V_\mathcal{A}, Q_\mathcal{D}\right] \leq 2^{-\gamma_{i^*}},$$

for all $x \in \{0,1\}^{p_i}$, which refers to the length of the I_i which can be of arbitrary length but we assume are a-priori fixed parameters of the samplers.

```
**Game** $\mathsf{GLEG}_{\mathsf{q}_\pi,i^*}(\mathcal{A},\mathcal{D})$
$\pi \xleftarrow{\$} \mathcal{P}_n$
**for** $j = 1,\ldots,\mathsf{q}_\mathcal{D}$ **do**
  $(\sigma_i, \mathcal{S}_i, \gamma_i, z_i) \xleftarrow{\$} \mathcal{D}^\pi$
  $\mathrm{I}_i \xleftarrow{\$} \mathcal{S}_i^\pi$
$V_\mathcal{A} \xleftarrow{\$} \mathcal{A}((\gamma_j, z_j)_{j\in[\mathsf{q}_\mathcal{D}]}, (\mathrm{I}_j)_{j\in[\mathsf{q}_\mathcal{D}]/\{i^*\}})$
**return** $((\mathrm{I}_1,\gamma_1,z_1),\ldots,(\mathrm{I}_{\mathsf{q}_\mathcal{D}},\gamma_{\mathsf{q}_\mathcal{D}},z_{\mathsf{q}_\mathcal{D}}), V_\mathcal{A}, Q_\mathcal{D})$
```

Figure 3.1: The game GLEG.

Definition 3.1.2 (Originally from [29] but as amended in [33]). A distribution sampler $\mathcal{D}$ as defined above in Definition 3.1.1 is $(\mathsf{q}_\mathcal{D}, \mathsf{q}_\pi)$-legitimate, if, for every adversary $\mathcal{A}$ making q_π queries, every $i^* \in [\mathsf{q}_\mathcal{D}]$, and for any possible values

$$(\mathrm{I}_j)_{j\neq i^*}, (\gamma_1, z_1), \ldots, (\gamma_{\mathsf{q}_\mathcal{D}}, z_{\mathsf{q}_\mathcal{D}}), V_\mathcal{A}, Q_\mathcal{D}$$

potentially output by the game $\mathsf{GLEG}_{\mathsf{q}_\mathcal{D},i^*}(\mathcal{A},\mathcal{D})$ with positive probability,

$$\Pr\left[\mathrm{I}_{i^*} = x \mid (\mathrm{I}_j)_{j\neq i^*}, (\gamma_1, z_1), \ldots, (\gamma_{\mathsf{q}_\mathcal{D}}, z_{\mathsf{q}_\mathcal{D}}), V_\mathcal{A}, Q_\mathcal{D}\right] \leq 2^{-\gamma_{i^*}},$$

for all $x \in \{0,1\}^{\ell_{i^*}}$, where the probability is conditioned on these particular values being output by the game.

3.2 Constructions

First we revisit the original sponge-based PRNG design of [18] as updated in [33]. The $\mathsf{sponge.prng}$ initialises a sponge in the usual way, with input materials being absorbed via the outer state of the sponge and outputs being read also from the outer state. There is an optional subroutine called $\mathsf{p.forget}$ that zeroes the outer state after an output to make the generator forward secure. This introduces the need to permute the state *before* each output, and then a further t times after the output has been read from the outer state.

```
p.forget($s_0$)
for $k = 1$ to $t$ do
    $s_k \leftarrow \pi(s_{k-1})$
    $s_k \leftarrow (0^r \| \widehat{s}_k)$
return $s_t$
```

```
sponge.prng.next(seed, $s_i$)
$s_0 \leftarrow \pi(s_i)$
$r_{i+1} \leftarrow \bar{s}_0$
$s_{i+1} \leftarrow$ p.forget($s_0$)
$j \leftarrow 1$
return $(s_{i+1}, r_{i+1})$
```

```
sponge.prng.setup$^\pi$()
for $i = 0, \ldots, u-1$ do
    seed$_i \xleftarrow{\$} \{0,1\}^r$
seed $\leftarrow$ (seed$_0, \ldots,$ seed$_{u-1}$)
$j \leftarrow 1$
return seed
```

```
sponge.prng.refresh(seed, $s_i$, I)
$s_{i+1} \leftarrow \pi((\bar{s}_i \oplus \mathrm{I} \oplus \text{seed}_j) \| \widehat{s}_i)$
$j \leftarrow j + 1 \mod u$
return $s_{i+1}$
```

Figure 3.2: The algorithms describing the behaviour of the sponge.prng.

3.2.1 The Design of sponge.prng

The sponge.prng design originally defined in [18] (which is given in Section 2.5.3) with the additions from [33] including the addition of a seed, is defined as follows:

Definition 3.2.1. Let $n, r, c \in \mathbb{N}$, and let seed $\in \{0,1\}^{ur}$, t the number of times the outer state is zeroed with $u, t > 1$, then

$$\text{sponge.prng}^{\pi}_{u,t,n,r} := (\text{sponge.prng.setup}^{\pi}, \text{sponge.prng.refresh}^{\pi}, \text{sponge.prng.next}^{\pi}),$$

as described in Figure 3.2.

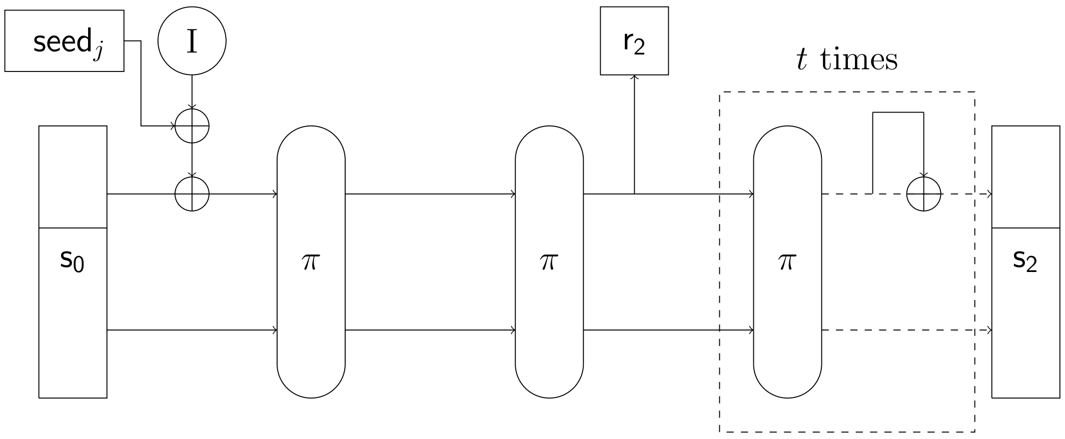


Figure 3.3: The sponge.prng in operation.

We improve the design of the next algorithm to ensure our design is more efficient, making a single call to the permutation π, compared with $1 + t$ calls in the p.forget procedure. This results in a design better suited for practical application, especially those that restrict the number of calls to π due to hardware or clock restraints, such as smart card usage. Since the p.forget procedure of the previous design calls the permutation $1+t$ times, with zeroing, it presents the problem of increased collisions in the state, something that is avoided by our design and thus our bound is mainly limited by the collision factor associated with the refresh procedure. This potentially makes our generator comparatively more secure, since the zeroing of the outer state in the p.forget procedure means a collision in the inner state leads to a full-state collision and thus an output cycle which is avoided in our design.

Our improved design can be seen in Figure 3.5 for further clarity. Although this design departs slightly from the sponge design, it can still be captured by the more generalised structure of the parazoa as defined in [3] which we explore in Section 3.6. Given access to the underlying permutation function, our design can be easily implemented.

```
Reverie.setup^π()
for $i = 0, \ldots, u-1$ do
    $\mathsf{seed}_i \xleftarrow{\$} \{0,1\}^r$
$\mathsf{seed} \leftarrow (\mathsf{seed}_0, \ldots, \mathsf{seed}_{u-1})$
$j \leftarrow 1$
return seed

Reverie.next(seed, $s_i$)
$r_{i+1} \leftarrow \bar{s}_i$
$s_{i+1} \leftarrow (\pi(s_i) \oplus (0^r \| \widehat{s}_i))$
return $(s_{i+1}, r_{i+1})$

Reverie.refresh(seed, $s_i$, I)
$s_{i+1} \leftarrow \pi((\bar{s}_i \oplus \mathrm{I} \oplus \mathsf{seed}_j) \| \widehat{s}_i)$
$j \leftarrow j + 1 \mod u$
return $s_{i+1}$
```

Figure 3.4: The algorithms describing the behaviour of Reverie.

3.2.2 The Design of Reverie

The design of Reverie was inspired by the Davies-Meyer compression function after identifying that so many calls to the underlying permutation in the sponge.prngp.forget procedure seemed very inefficient with room for improvement. We considered several different options, such as splitting the state and using one part to key a permutation for the other part and vice versa, but decided on the Davies-Meyer inspired option due to its simplicity.

Definition 3.2.2. Let $n, r, c \geq 1$ and $c := n - r, \ell = p = r$, together with $\pi \xleftarrow{\$} \mathcal{P}_n$, and seed $\mathsf{seed} \in \{0,1\}^{ur}$ with $u > 1$, then

$$\mathsf{Rev}^{\pi}_{u,n,r} := (\mathsf{Reverie.setup}^{\pi}, \mathsf{Reverie.refresh}^{\pi}, \mathsf{Reverie.next}^{\pi}),$$

as described in Figure 3.4.

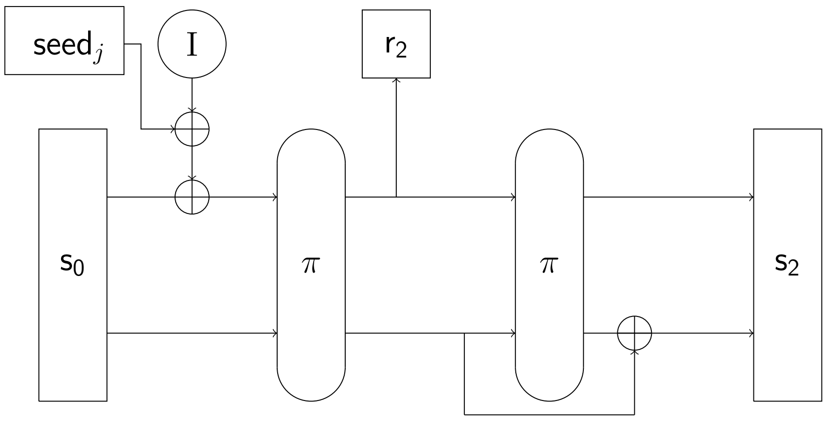


Figure 3.5: The Reverie PRNG in operation.

3.2.3 Differentiability of the Construction

Unfortunately the indifferentiability proof of the sponge construction does not apply to Reverie for several reasons; one is that Reverie is not a sponge and second, a random oracle does not capture the behaviour of a PRNG nor the adversarial access afforded to an adversary in the PRNG security model. Even if $\mathcal{RO}$ is replaced with some notion of an ideal PRNG, the existence of the public random permutation prevents an indifferentiability proof from being possible. We also note that the paper by Ristenpart, Shacham and Shrimpton [51] implies that utilising an indifferentiability result would not be possible in proving the robustness of the construction. This is in part due to the security notions having multiple disjoint adversarial stages.

3.3 Security Notions in the Ideal Permutation Model

This section defines the notion of robustness originally from [29], but augmented as in [33], to allow for the publicly available random permutation. Robustness is the strongest security notion of the security model. We also include definitions of two weaker notions of security; preserving and recovering security, which together imply that a PRNG fulfils the requirements of robustness.

As per the definitions of [29], γ^* refers to a minimal "fresh" entropy in the PRNG system when security should be expected but below which is assumed to be compromised. Minimising γ^* corresponds to a stronger security guarantee.

An adversary is modelled using a pair $(\mathcal{A}, \mathcal{D})$, where $\mathcal{A}$ is the actual q_π-query adversary and $\mathcal{D}$ is a $(\mathsf{q}_\mathcal{D}, \mathsf{q}_\pi)$-legitimate distribution sampler. The adversary $\mathcal{A}$'s goal is to determine a challenge bit b picked during the initialise procedure; this procedure also returns seed to the adversary.

Definition 3.3.1. A PRNG with input G is called $((\mathsf{q}_\pi, \mathsf{q}_\mathcal{D}, \mathsf{q}_\mathsf{R}, \mathsf{q}_\mathsf{S}), \gamma^*, \epsilon_{\mathsf{rob}})$-robust $(\mathsf{rob}_\mathsf{G}^{\gamma^*})$ if for any adversary $\mathcal{A}$ making at most q_π queries to $\pi^\pm$, making at most $\mathsf{q}_\mathcal{D}$ calls to $\mathcal{D}-\mathsf{refresh}$, q_R calls to $\mathsf{next\text{-}ror}/\mathsf{get\text{-}next}$ and q_S calls to $\mathsf{get\text{-}state}/\mathsf{set\text{-}state}$ and any legitimate distribution sampler $\mathcal{D}$, the advantage of any adversary in the robustness game is at most ϵ_{rob}, which is defined in Equation (3.3.1). The adversary

```
Procedure: Initialise ()
$\boxed{\pi \xleftarrow{\$} \mathcal{P}_n}$
seed $\xleftarrow{\$}$ $\boxed{\mathsf{setup}^{\pi}}$
$s_0 \xleftarrow{\$} \{0,1\}^n$
$\sigma \leftarrow \perp$
corrupt $\leftarrow$ false
$e \leftarrow n$
b $\xleftarrow{\$} \{0,1\}$
return seed
```

```
Procedure: Finalise (b*)
if b = b* then
    return 1
else
    return 0
```

```
Oracle: $\pi(x)$
return $\pi(x)$
```

```
Oracle: $\pi^{-1}(x)$
return $\pi^{-1}(x)$
```

```
Procedure: get-state ()
$e \leftarrow 0$
corrupt $\leftarrow$ true
return $s_i$
```

```
Procedure: set-state ($s^*$)
$e \leftarrow 0$
corrupt $\leftarrow$ true
$s_i \leftarrow s^*$
```

```
Procedure: next-ror ()
$(s_{i+1}, r_0) \leftarrow \boxed{\mathsf{next}^{\pi}}(\mathsf{seed}, s_i)$
$r_1 \xleftarrow{\$} \{0,1\}^r$
if corrupt = true
    $e \leftarrow 0$
    return $r_0$
else
    return $r_{\mathsf{b}}$
```

```
Procedure: get-next ()
$(s_{i+1}, r_{i+1}) \leftarrow \boxed{\mathsf{next}^{\pi}}(\mathsf{seed}, s_i)$
if corrupt = true then
    $e \leftarrow 0$
return $r_{i+1}$
```

```
Procedure: $\mathcal{D}$−refresh()
$(\sigma, \mathrm{I}, \gamma, z) \xleftarrow{\$} \boxed{\mathcal{D}^{\pi}}(\sigma)$
$s_{i+1} \leftarrow \boxed{\mathsf{refresh}^{\pi}}(\mathsf{seed}, s_i, \mathrm{I})$
$e \leftarrow e + \gamma$
if $e \geq \gamma^*$ then
    corrupt $\leftarrow$ false
return $(\gamma, z)$
```

Figure 3.6: The updated security procedures, updated from the original definitions given in [29]. Boxed items indicate changes.

$\mathcal{A}$ has access to a subset of the following oracles, dependent on the security game that it is playing; the full set is available in $\mathsf{rob}_{\mathsf{G}}^{\gamma^*}(\mathcal{A},\mathcal{D})$. We say that an adversarial pair $(\mathcal{A},\mathcal{D})$ playing the robustness game as described in Figure 3.6, with a PRNG G, have advantage

$$\mathsf{Adv}_{\mathsf{G}}^{\gamma^*-\mathsf{rob}}(\mathcal{A},\mathcal{D}) := \left| 2\Pr\left[\mathsf{rob}_{\mathsf{G}}^{\gamma^*}(\mathcal{A},\mathcal{D}) \Rightarrow 1\right] - 1 \right| \leq \epsilon_{\mathsf{rob}}. \tag{3.3.1}$$

Next, we define two further security notions: preserving security and recovering security. If a PRNG satisfies both these notions then, by Theorem 1 of [29] (updated in the IPM in [33, Theorem 4]), the generator in question satisfies the robustness security notion under the corresponding parameters.

```
$\mathsf{Preserve}_{\mathsf{G}}(\mathcal{A})$
$\boxed{\pi \xleftarrow{\$} \mathcal{P}_n}$, $\mathsf{seed} \xleftarrow{\$} \boxed{\mathsf{setup}^{\pi}()}$, $\mathsf{b} \xleftarrow{\$} \{0,1\}$, $s_0 \xleftarrow{\$} \{0,1\}^n$
$(\mathrm{I}_1, \ldots, \mathrm{I}_d) \leftarrow \boxed{\mathcal{A}^{\pi}}(\mathsf{seed})$
for $j = 1, \ldots, d$ do
    $s_j \leftarrow \boxed{\mathsf{refresh}^{\pi}}(\mathsf{seed}, s_{j-1}, \mathrm{I}_j)$
$(s_0, r_0) \leftarrow \boxed{\mathsf{next}^{\pi}}(\mathsf{seed}, s_d)$
$(s_1, r_1) \xleftarrow{\$} \{0,1\}^n \times \{0,1\}^r$
$\mathsf{b}^* \leftarrow \boxed{\mathcal{A}^{\pi}}(s_{\mathsf{b}}, r_{\mathsf{b}})$
return $\mathsf{b} == \mathsf{b}^*$
```

Figure 3.7: The security game for preserving security, updated from the original definitions given in [29, Definition 4]. Boxed items indicate changes.

3.3.1 Preserving Security

Informally, preserving security states that if the state of a generator starts uncompromised, and is refreshed using compromised input, then the next output and resulting state are still indistinguishable from random.

Definition 3.3.2. A PRNG with input is said to have $(\mathsf{q}_\pi, \epsilon_{\mathsf{p}})$-preserving security if the advantage of any adversary $\mathcal{A}$ making at most q_π queries to $\pi^{\pm}$ in the game given in Figure 3.7 is at most ϵ_{p}, where the advantage is defined to be

$$\mathsf{Adv}_{\mathsf{G}}^{\mathsf{pres}}(\mathcal{A}) := |2\Pr\left[\mathsf{Preserve}_{\mathsf{G}}(\mathcal{A}) \Rightarrow 1\right] - 1| \leq \epsilon_{\mathsf{p}}.$$

```
$\underline{\mathsf{Recover}_{\mathsf{G}}^{(\gamma^*,\mathsf{q}_\pi)}(\mathcal{A},\mathcal{D})}$
$\boxed{\pi \xleftarrow{\$} \mathcal{P}_n}, \mathsf{seed} \xleftarrow{\$} \boxed{\mathsf{setup}^\pi()}, \mathsf{b} \xleftarrow{\$} \{0,1\}, \sigma_0 \leftarrow \perp$
for $k = 1, \ldots, \mathsf{q}_\mathcal{D}$ do
    $(\sigma_k, \mathrm{I}_k, \gamma_k, z_k) \leftarrow \boxed{\mathcal{D}^\pi}(\sigma_{k-1})$
$k \leftarrow 0$
$(s_0, d) \leftarrow \boxed{\mathcal{A}^{\pi,\mathsf{get\text{-}refresh}}}(\gamma_1, \ldots, \gamma_{\mathsf{q}_\mathcal{D}}, z_1, \ldots, z_{\mathsf{q}_\mathcal{D}}, \mathsf{seed})$
if $k + d > \mathsf{q}_\mathcal{D}$ then return $\perp$
else
if $\sum_{j=k+1}^{k+d} \gamma_j < \gamma^*$ then return $\perp$
else
for $j = 1, \ldots, d$ do
    $s_j \leftarrow \boxed{\mathsf{refresh}^\pi}(s_{j-1}, \mathrm{I}_{k+j}, \mathsf{seed})$
$(s_0, r_0) \leftarrow \boxed{\mathsf{next}^\pi}(\mathsf{seed}, s_d)$
$(s_1, r_1) \xleftarrow{\$} \{0,1\}^n \times \{0,1\}^r$
$\mathsf{b}^* \leftarrow \boxed{\mathcal{A}^\pi}((s_\mathsf{b}, r_\mathsf{b}), \mathrm{I}_{k+d+1}, \ldots, \mathrm{I}_{\mathsf{q}_\mathcal{D}})$
return $\mathsf{b} == \mathsf{b}^*$

$\underline{\text{Oracle } \mathsf{get\text{-}refresh}\ ()}$
$k \leftarrow k + 1$
return $\mathrm{I}_k$
```

Figure 3.8: The security game for recovering security, updated from the original definitions given in [29, Definitions 3]. Boxed items indicate changes.

3.3.2 Recovering Security

Informally, recovering security implies that if a PRNG is compromised, inserting enough random entropy to refresh the internal state will ensure that the next output and state will be indistinguishable from random.

Definition 3.3.3. A PRNG with input has $(\mathsf{q}_\pi, \mathsf{q}_\mathcal{D}, \gamma^*, \epsilon_\mathsf{r})$-recovering security if the advantage of any adversary $\mathcal{A}$ making at most $\mathsf{q}_\mathcal{D}$ queries to $\pi^\pm$ and distribution sampler $\mathcal{D}$, making at most $Q(\mathsf{q}_\mathcal{D})$ queries to $\pi^\pm$, in the following game with $\gamma^* > 0$ is at most ϵ_r where advantage is defined as

$$\mathsf{Adv}_\mathsf{G}^{(\gamma^*,\mathsf{q}_\mathcal{D})-\mathsf{rec}}(\mathcal{A},\mathcal{D}) := \left| 2\Pr\left[\mathsf{Recover}_\mathsf{G}^{(\gamma^*,\mathsf{q}_\mathcal{D})} \Rightarrow 1\right] - 1 \right| \leq \epsilon_\mathsf{r}.$$

3.4 Security Proofs

This section consists of the security proofs of $\mathsf{Reverie}$; the approach is to analyse the security of the next function as a PRG, which can then be applied in both preserving and recovering security. We then focus on the preserving and recovering security games, making use of the IPM composition theorem [33, Theorem 4].

Theorem 3.4.1.
For Reverie $= \mathsf{Rev}^{\pi}_{u,n,r}$ *as defined in Definition 3.2.2, let* $\gamma^* > 0$, *let* $\mathcal{D}$ *be a* $(\mathsf{q}_{\mathcal{D}}, \mathsf{q}_{\pi})$*-legitimate distribution sampler, let* $\bar{q}_{\pi} := \mathsf{q}_{\pi} + Q(\mathsf{q}_{\mathcal{D}})$ *and* $\widehat{q} := \bar{q}_{\pi} + \mathsf{q}_{\mathsf{R}} + \mathsf{q}_{\mathcal{D}} d$. *Then* $\mathsf{Rev}^{\pi}_{u,n,r}$ *is* $((\mathsf{q}_{\pi}, \mathsf{q}_{\mathcal{D}}, \mathsf{q}_{\mathsf{R}}, \mathsf{q}_{\mathsf{S}}), \gamma^*, \epsilon_{\mathsf{r}})$*-robust, for* ϵ_{rob} *as defined in the following equation:*

$$\mathsf{Adv}^{\gamma^*\text{-}\mathsf{rob}}_{\mathsf{Rev}^{\pi}_{\mathsf{u,n,r}}}(\mathcal{A}, \mathcal{D}) \leq \mathsf{q}_{\mathsf{R}} \cdot \left(\frac{\bar{q}_{\pi} + 1}{2^{\gamma^*}} + \frac{Q(\mathsf{q}_{\mathcal{D}})}{2^{ur}} + \frac{7(\widehat{q}^2 + 1) + 29\widehat{q}}{2^{c-1}} \right.$$
$$\left. + \frac{(2d^2 + 3)\widehat{q} + d(3d + 2d)}{2^n} \right).$$

Proof. The theorem is the result of the preserving and recovering security bounds in Lemmas 3.4.11 and 3.4.13 respectively, combined by [33, Theorem 4], stated in Theorem 2.3.10. □

Lemma 3.4.2 (PRG security of the next function)**.**
Let U_x *be the uniform distribution over* x*-bit strings, let* next *be as defined in Section 3.2.2, let* $s_0 \xleftarrow{\$} \{0,1\}^n$, *then for any* q_{π}*-query adversary* $\mathcal{A}$,

$$\epsilon_{\mathsf{PRG}} := \mathsf{Adv}^{\mathsf{dist}}_{\mathcal{A}}(\mathsf{next}(\mathsf{U}_n), (\mathsf{U}_n, \mathsf{U}_r)) \leq \left(2 - \frac{1}{2^r}\right) \frac{\mathsf{q}_{\pi}}{2^{c-1}} + \frac{3\mathsf{q}_{\pi}}{2^{c-1}}$$
$$= \left(5 - \frac{1}{2^r}\right) \frac{\mathsf{q}_{\pi}}{2^{c-1}}.$$

Proof outline: Distinguishing between $\mathsf{next}(s_0)$ and random output $(s_1, r_1) \xleftarrow{\$} \{0,1\}^n \times \{0,1\}^r$ naively, it seems like the adversary's only option is to guess the inner state of the secret initial state, by either a direct forward query to π or by an indirect guess that would reveal a candidate for this inner state through a query to π^{-1}.

The proof proceeds by showing that this is in fact the optimal strategy. Since there are two parts to the challenge, the logical approach is to split the proof into first proving that one part of the challenge can be replaced with random, before approaching the remaining part of the challenge.

We note that unlike [33], the next function requires a uniformly random state; the difference is made up for in a game jump in the proof, but allows us to avoid an additional call to π, as is required in [33]. This step can be reinstated at the cost of a single additional call to π.

Proof. The formal proof proceeds by first defining three versions of the next algorithm in Figure 3.9.

These algorithms are set up so that on input $s_0 \xleftarrow{\$} \{0,1\}^n$, next_0 is precisely the next function on input s_0, while next_2 has the same distribution as $(\mathsf{U}_n, \mathsf{U}_r)$. Lastly, next_1^π will be used as a hybrid game. Thus, by the triangle inequality,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{dist}}_{\mathcal{A}}(\mathsf{next}(s_0), (\mathsf{U}_n, \mathsf{U}_r)) \leq & \mathsf{Adv}^{\mathsf{dist}}_{\mathcal{A}}(\mathsf{next}_0^\pi(s_0), \mathsf{next}_1^\pi(s_0)) \\ & + \mathsf{Adv}^{\mathsf{dist}}_{\mathcal{A}}(\mathsf{next}_1^\pi(s_0), \mathsf{next}_2(s_0)).\end{aligned}$$

What follows is to prove the bound using the H-coefficient technique. As described in Section 2.7.2, we assume that $\mathcal{A}$ is deterministic and makes q_π non-repeating queries to the permutation π, denoted as

$$\tau_{\mathcal{A}} := (x_1, y_1, z_1), \ldots, (x_{\mathsf{q}_\pi}, y_{\mathsf{q}_\pi}, z_{\mathsf{q}_\pi}),$$

```
Algorithm $\mathsf{next}_0^\pi(s_0)$
$t \leftarrow \bar{s}$
$t \leftarrow \pi(s_0)$
$s \leftarrow t \oplus (0^r \| \widehat{s}_0)$
return $(s, t)$
```

```
Algorithm $\mathsf{next}_1^\pi(s_0)$
$t \xleftarrow{\$} \{0,1\}^r$
$t \leftarrow \pi(s_0)$
$s \leftarrow t \oplus (0^r \| \widehat{s}_0)$
return $(s, t)$
```

```
Algorithm $\mathsf{next}_2(s_0)$
$t \xleftarrow{\$} \{0,1\}^r$
$s \xleftarrow{\$} \{0,1\}^n$
return $(s, t)$
```

Figure 3.9: The algorithms $\mathsf{next}_0^\pi, \mathsf{next}_1^\pi, \mathsf{next}_2$ used in proving the security of the next function.

where $\forall i \in [1, \dots, \mathsf{q}_\pi]$,

$$y_i = \pi(x_i),$$
$$z_i = y_i \oplus (0^r \| \widehat{x}_i).$$

In addition to the challenge, the adversary in this distinguishing game is also given several other pieces of information at the end of the game, after all queries to π have been made, but before the adversary must output her decision. Formally, $\mathcal{A}$ is given $\widehat{s}_0$ and $t' := (\bar{s} \| (\widehat{s}_0 \oplus \widehat{s}))$ which it can compute for itself but is given for clarity. This completes the definition of a transcript for these experiments,

$$\tau := ((x_1, y_1, z_1), \dots, (x_{\mathsf{q}_\pi}, y_{\mathsf{q}_\pi}, z_{\mathsf{q}_\pi}), \widehat{s}_0, t', (s, t)). \tag{3.4.1}$$

We say a transcript τ is compatible with $\mathsf{next}_0^\pi(s_0)$ if it can be output in the experiment where $\mathcal{A}$ receives $\mathsf{next}_0^\pi(s_0)$. Since $\mathsf{next}_1^\pi(s_0)$ and $\mathsf{next}_2(s_0)$ differ only by replacing real output with random, it is clear that if a transcript is compatible with $\mathsf{next}_0^\pi(s_0)$ then it is compatible with $\mathsf{next}_1^\pi(s_0)$ and $\mathsf{next}_2(s_0)$.

What follows is bounding the probability of different transcripts from each experiment.

Lemma 3.4.3.
For the experiments $\mathsf{next}_0^\pi(s_0), \mathsf{next}_1^\pi(s_0)$ *as described in Figure 3.9,*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{dist}}(\mathsf{next}_0^\pi(s_0), \mathsf{next}_1^\pi(s_0)) \leq \left(2 - \frac{1}{2^r}\right) \frac{\mathsf{q}_\pi}{2^{c-1}} + 0 = \left(2 - \frac{1}{2^r}\right) \frac{\mathsf{q}_\pi}{2^{c-1}}.$$

Proof. First we define the bad transcripts for this pair of experiments:

Definition 3.4.4 (Bad transcripts $\mathcal{T}_{\mathsf{Bad}}$ for $(\mathsf{next}_0^\pi(s_0)\mathsf{next}_1^\pi(s_0))$)**.** A compatible transcript as in Equation (3.4.1), is called a bad transcript if either of the following occur:

$$\text{State Collision (SC): } \exists j \in [\mathsf{q}_\pi] \text{ such that } x_j = (t \| \widehat{s}_0),$$
$$\text{Image Collision (IC): } \exists j \in [\mathsf{q}_\pi] \text{ such that } y_j = t'.$$

The set of bad transcripts is denoted $\mathcal{T}_{\mathsf{Bad}}$.

Let X_0, Y_0 be the random variables outputting transcripts that describe when $\mathcal{A}$ interacts with $\mathsf{next}_0^\pi(s_0)$ and $\mathsf{next}_1^\pi(s_0)$ respectively.

Lemma 3.4.5.
For an adversary making no more than $\mathsf{q}_\pi \leq 2^{c-1}$ *queries to an oracle in the experiment* $\mathsf{next}_1(s_0)$,

$$\Pr\left[Y_0 \in \mathcal{T}_{\mathsf{Bad}}\right] \leq \left(2 - \frac{1}{2^r}\right)\frac{\mathsf{q}_\pi}{2^{c-1}}.$$

Proof. Note that if $Y_0 \in \mathcal{T}_{\mathsf{Bad}}$ then SC $\vee$ IC must occur.

$$\Pr\left[Y_0 \in \mathcal{T}_{\mathsf{Bad}}\right] \leq \Pr\left[\mathrm{SC}\right] + \Pr\left[\mathrm{IC} \mid \neg\mathrm{SC}\right].$$

The first probability is relatively easy to bound:

$$\Pr\left[\mathrm{SC}\right] \leq \frac{\mathsf{q}_\pi}{2^{c-1}}. \tag{3.4.2}$$

Since the adversary is given t at the start of the game and s_0 is uniformly distributed over all the 2^c n-bit strings with outer bits equal to t, and recalling that $\mathsf{q}_\pi \leq 2^{c-1}$, the probability that $\mathcal{A}$'s i-th query is of the form $((t\|\widehat{s}_0), y_i, z_i)$ is $\frac{1}{2^c-i+1}$. More formally, let $\Pr\left[win_i\right] := \Pr\left[x_i = (t\|\widehat{s}_0)\right]$, then

$$\begin{aligned}\Pr\left[win\right] &\leq \sum_{i=1}^{\mathsf{q}_\pi} \Pr\left[win_i\right] = \sum_{i=1}^{\mathsf{q}_\pi} \frac{1}{2^c - i + 1} \\ &\leq \sum_{i=1}^{\mathsf{q}_\pi} \frac{1}{2^c - 2^{c-1}} = \frac{\mathsf{q}_\pi}{2^{c-1}}.\end{aligned}$$

The second, since SC has not occurred, must be where the adversary is interacting with $\mathsf{next}_1^\pi(s_0)$, where t was chosen uniformly at random from r-bit strings, and as such, was not used to produce s. There is the possibility that the randomly chosen t matches the real value of $\bar{s}_0$ which is reflected in the factor of $\left(1 - \frac{1}{2^r}\right)$ in Equation (3.4.3).

The second probability is similar, in that the adversary has knowledge of $\bar{s}$, with $(\bar{s}\|(\widehat{s}_0 \oplus \widehat{s}))$ uniformly distributed over all the 2^c n-bit strings with outer bits equal to $\bar{s}$. It is also assumed that a SC has not occurred, meaning nothing beyond $\widehat{s}_0$ is known about s_0. Then, similarly to above,

$$\Pr\left[\mathrm{IC} \mid \neg\mathrm{SC}\right] \leq \left(1 - \frac{1}{2^r}\right)\frac{\mathsf{q}_\pi}{2^{c-1}}. \tag{3.4.3}$$

Equation (3.4.3), together with Equation (3.4.2), complete the lemma. □

Lemma 3.4.6.
For all compatible transcripts $\tau \in \mathcal{T}_{\mathsf{Good}}$,

$$\Pr[X_0 = \tau] = \Pr[Y_0 = \tau].$$

Proof. For all $\tau \in \mathcal{T}_{\mathsf{Good}}$ (and for $\pi \xleftarrow{\$} \mathcal{P}_n$),

$$\begin{aligned}\Pr[X_0 = \tau] = \\ &\Pr[\forall i \in [\mathsf{q}_\pi], \pi(x_i) = y_i] \cdot \Pr[\pi(s_0) = (\bar{s} \| (\widehat{s}_0 \oplus \widehat{s})) \mid \neg\mathrm{SC} \vee \neg\mathrm{IC}] \\ &= \frac{1}{2^r} 2^r \frac{(2^n - \mathsf{q}_\pi - 1)!}{2^n!} = \Pr[Y_1 = \tau].\end{aligned}$$

□

Putting Lemmas 3.4.5 and 3.4.6 together yields the result. □

Next, we prove the following:

Lemma 3.4.7.
For the experiments $\mathsf{next}_1^\pi(s_0), \mathsf{next}_2(s_0)$ as described in Figure 3.9 and by Theorem 2.7.7,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{dist}}(\mathsf{next}_1^\pi(s_0), \mathsf{next}_2(s_0)) \leq \frac{3\mathsf{q}_\pi}{2^{c-1}} + 0 = \frac{3\mathsf{q}_\pi}{2^{c-1}}.$$

Proof. This time the transcript is slightly different, in that the adversary is given the entire s_0 at the end of her queries to π, so

$$\tau := ((x_1, y_1, z_1), \ldots, (x_{\mathsf{q}_\pi}, y_{\mathsf{q}_\pi}, z_{\mathsf{q}_\pi}), s_0, t', (s, t)).$$

Comparing the distributions of these two experiments yields one more bad event, along with a modified state collision and unchanged image collision:

Definition 3.4.8 (Bad transcripts $\mathcal{T}_{\mathsf{Bad}}$ for $(\mathsf{next}_1^\pi(s_0), \mathsf{next}_2(s_0))$)**.** A compatible transcript as above, is called a bad transcript if any of the following occur:

$$\begin{aligned}\text{State Collision (SC): } &\exists j \in [\mathsf{q}_\pi] \text{ such that } x_j = s_0, \\ \text{Image Collision (IC): } &\exists j \in [\mathsf{q}_\pi] \text{ such that } y_j = t', \\ \text{Inversion (IN): } &\exists j \in [\mathsf{q}_\pi] \text{ such that } z_j = s.\end{aligned}$$

The set of bad transcripts is denoted $\mathcal{T}_{\mathsf{Bad}}$.

Let X_1, Y_1 be the random variables outputting transcripts that describe when $\mathcal{A}$ interacts with $\mathsf{next}_1^\pi(s_0)$ and $\mathsf{next}_2(s_0)$ respectively.

Lemma 3.4.9.
For an adversary making no more than $\mathsf{q}_\pi \leq 2^{c-1}$ *queries to an oracle in the experiment* $\mathsf{next}_2(s_0)$*,*

$$\Pr\left[Y_1 \in \mathcal{T}_{\mathsf{Bad}}\right] \leq \frac{\mathsf{q}_\pi}{2^{n-1}} + \left(2 - \frac{1}{2^r}\right)\frac{\mathsf{q}_\pi}{2^{c-1}} = \frac{2\mathsf{q}_\pi}{2^{c-1}}.$$

Proof. Note that if $Y_1 \in \mathcal{T}_{\mathsf{Bad}}$ then SC $\vee$ IC $\vee$ IN must occur, and thus:

$$\Pr\left[Y_1 \in \mathcal{T}_{\mathsf{Bad}}\right] \leq \Pr\left[\mathrm{SC}\right] + \Pr\left[\mathrm{IC} \mid \mathrm{SC}\right] + \Pr\left[\mathrm{IN} \mid \neg\mathrm{SC} \wedge \neg\mathrm{IC}\right].$$

The first probability is similar to before, but this time the adversary knows that $\bar{s}$ (with high probability) was not queried to π to produce the challenge. This results in the following:

$$\Pr\left[\mathrm{SC}\right] \leq \frac{\mathsf{q}_\pi}{2^{n-1}}.$$

The second probability is similar to the case where an IC occurs in a transcript in either $\mathsf{next}_0^\pi(s_0)$ or $\mathsf{next}_1^\pi(s_0)$. Once again since $\widehat{s}_0$ is uniformly distributed over $\{0,1\}^c$, the probability that any of the adversary's queries $(x_i) = y_i$ or $\pi^{-1}(y_i) = x_i$ is such that $y_i = (\bar{s}\|\widehat{s} \oplus \widehat{s}_0)$ is at most $\frac{1}{2^{c}-i+1}$ resulting in the bound $\frac{\mathsf{q}_\pi}{2^{c-1}}$. It is also assumed that a SC has not occurred, meaning nothing beyond $\widehat{s}_0$ is known about s_0. Thus,

$$\Pr\left[\mathrm{IC} \mid \neg\mathrm{SC}\right] \leq \left(1 - \frac{1}{2^r}\right)\frac{\mathsf{q}_\pi}{2^{c-1}}.$$

Lastly, if neither a SC or IC has occurred, the probability of an IN can be expressed as

$$\Pr\left[\pi^{-1}(\bar{s}\|\widehat{y}_i) = (\widehat{x}_i\|(\widehat{y}_i \oplus \widehat{s}))\right],$$

which again is bounded by $\frac{\mathsf{q}_\pi}{2^{c-1}}$ and together with the other events, yields the desired bound. □

Lemma 3.4.10.
For all compatible transcripts $\tau \in \mathcal{T}_{\mathsf{Good}}$,

$$\Pr\left[\Omega_{X_1} = \tau\right] = \Pr\left[\Omega_{Y_1} = \tau\right].$$

For all $\tau \in \mathcal{T}_{\mathsf{Good}}$ (and for $\pi \xleftarrow{\$} \mathcal{P}_n$),

Proof.

$$\begin{aligned}\Pr\left[X_1 = \tau\right] &= \Pr\left[\forall i \in [\mathsf{q}_\pi], \pi(x_i) = y_i\right] \cdot \Pr\left[\pi(s_0) = (\bar{s} \| (\widehat{s}_0 \oplus \widehat{s})) \mid \neg\mathrm{SC} \vee \neg\mathrm{IC} \vee \neg\mathrm{IN}\right] \\ &= \frac{(2^n - \mathsf{q}_\pi - 1)!}{2^n!} = \frac{(2^n - \mathsf{q}_\pi)!}{2^n} \cdot \frac{1}{2^n - \mathsf{q}_\pi} = \Pr\left[Y_1 = \tau\right].\end{aligned}$$

□

Putting Lemmas 3.4.9 and 3.4.10 together yields the result. □

Finally, these two lemmas complete the proof of the security of next. □

3.4.1 Preserving Security

Now that we have this tool, we can prove the following:

Lemma 3.4.11.
Given $\mathsf{Reverie}$ as defined in Section 3.2.2, and with ϵ_{PRG} as above, then for every q_π-query adversary $\mathcal{A}$ playing the preserving security game defined in Definition 3.3.2 with d adversarial refresh inputs, we have

$$\begin{aligned}\mathsf{Adv}^{\mathsf{pres}}_{\mathsf{Rev}^{\pi}_{\mathsf{u,n,r}}}(\mathcal{A}) &\leq \epsilon_{\mathsf{PRG}}(\mathsf{q}_\pi) + \frac{\mathsf{q}_\pi{}' + d}{2^n} + \frac{(d+1)(2\mathsf{q}_\pi{}' + d)}{2^n} \\ &\leq \frac{5\mathsf{q}_\pi}{2^{c-1}} + \frac{(2d+3)\mathsf{q}_\pi + d(d+2)}{2^n},\end{aligned}$$

for $\mathsf{q}_\pi{}' := |\tau'_{\mathcal{A}}| \leq \mathsf{q}_\pi$.

Proof outline The proof relies on proving that for a random secret initial state s_0, the resulting state s_d will look random and thus, by our previous analysis of the next function, the challenge output will also be random.

Proof. Formally, we adapt the preserving security game, so that the intermediate state s_d is chosen uniformly at random rather than calculated using the adversarial inputs.

Let $\mathcal{A}$ be the adversary playing in the preserving security game. Define $\tau'_{\mathcal{A}}$ to be as in Equation (3.4.1); the set of adversarial queries, but, restricted to only those made in the first part of the game, before the adversary has submitted her inputs and such that $|\tau'_{\mathcal{A}}| := \mathsf{q}_\pi{}' \leq \mathsf{q}_\pi$. Let $\mathrm{I}_1, \ldots, \mathrm{I}_d$ be the r-bit adversarial refresh inputs.

Let $\mathsf{Preserve}_{\mathsf{Rev}^\pi}$ be the real world preserving security game as defined in Definition 3.3.2 with the defined algorithms of $\mathsf{Reverie}$ and chosen permutation π. Let $\mathsf{Preserve}'_{\mathsf{Rev}^\pi}$ be identical to $\mathsf{Preserve}_{\mathsf{Rev}^\pi}$ except s_d is replaced with $s_d \xleftarrow{\$} \{0,1\}^n$.

We now aim to prove in two parts that, in the real world case, the first two games act the same with a small bound while in the ideal world case, they are identical. Following this, what remains is to prove that the advantage of an adversary in distinguishing the ideal world from the real world in $\mathsf{Preserve}'$ is precisely the security bound of the next function from Lemma 3.4.2. For clarity, we say that $\mathsf{Preserve}_{\mathsf{Rev}^\pi}(\mathcal{A}) \to 1$ means the adversary outputs 1 as her guess of b.

Lemma 3.4.12.
For $\mathsf{Preserve}_{\mathsf{Rev}^\pi}$ and $\mathsf{Preserve}'_{\mathsf{Rev}^\pi}$ where the former is as described in Definition 3.3.2 and with the latter identical except $s_d \xleftarrow{\$} \{0,1\}^n$,

$$\left|\Pr\left[\mathsf{Preserve}^{\mathcal{A}}_{\mathsf{Rev}^\pi} \to 1 \mid b = 0\right] - \Pr\left[\mathsf{Preserve}'^{\mathcal{A}}_{\mathsf{Rev}^\pi} \to 1 \mid b = 0\right]\right| \leq \frac{\mathsf{q}_\pi{}' + d}{2^n} + \frac{(d+1)(2\mathsf{q}_\pi{}' + d)}{2^n}.$$

Proof. To begin, we note that $s_0 \overset{\$}{\leftarrow} \{0,1\}^n$, and is not revealed to the adversary. With this in mind, using lazy sampling of the permutation π (each new output is generated ad-hoc depending on what has been previously queried and output), we have

$$\Pr\left[\exists i \in [\mathsf{q}_\pi{}'] s.t. x_i = s_0 \oplus ((\mathrm{I}_1 \oplus \mathsf{seed}_1)\|0^c)\right] \leq \frac{\mathsf{q}_\pi{}'}{2^{n-1}}.$$

Provided that this does not happen, the first intermediate state of the adversarial refreshes will be an unassigned value s_1 which will be uniformly chosen over the remaining $2^n - \mathsf{q}_\pi{}'$ unassigned values of π, and thus the probability that the next call to π will be on an already assigned value will be $\frac{\mathsf{q}_\pi{}'+1}{2^{n-1}}$. Iterating this method, we obtain:

$$\frac{\mathsf{q}_\pi{}'}{2^{n-1}} + \frac{\mathsf{q}_\pi{}'+1}{2^{n-1}} + \cdots + \frac{\mathsf{q}_\pi{}'+d}{2^{n-1}} = \frac{d(2\mathsf{q}_\pi{}' + (d+1))}{2^n}.$$

So, with probability $1 - \frac{\mathsf{q}_\pi{}'}{2^{n-1}} + \frac{d(2\mathsf{q}_\pi{}'+(d+1)}{2^n} = 1 - \frac{(d+1)(2\mathsf{q}_\pi{}'+d)}{2^n}$, the resulting state s_d after the adversarial refreshes will be the result of π called on an unassigned state. Then s_d will be chosen uniformly from the remaining $2^n - \mathsf{q}_\pi{}' - d$ unassigned values.

Finally, this implies the statistical distance between s_d in $\mathsf{Preserve}_{\mathsf{Rev}^\pi}$ and s_d in $\mathsf{Preserve}'_{\mathsf{Rev}^\pi}$ is at most $\frac{\mathsf{q}_\pi{}'+d}{2^n}$, which, together with the previous probability, yields the result. □

Next, construct an adversary $\mathcal{A}'$ that runs $\mathcal{A}$ and simulates the $\mathsf{Preserve}'$ game while inserting its own challenge and outputting the same bit as $\mathcal{A}$, which yields

$$\begin{aligned}\Big|\Pr\Big[\mathsf{Preserve}^{\mathcal{A}}_{\mathsf{Rev}^\pi} \to 1 \mid b=0\Big] - \Pr\Big[\mathsf{Preserve}'^{\mathcal{A}}_{\mathsf{Rev}^\pi} \to 1 \mid b=1\Big]\Big| \\ \leq \mathsf{Adv}^{\mathsf{dist}}_{\mathcal{A}'}(\mathsf{next}^\pi(\mathsf{U}_n), (\mathsf{U}_n, \mathsf{U}_r)).\end{aligned}$$

This, together with Lemma 3.4.12, completes the proof. □

3.4.2 Recovering Security

Thanks to the result of [33], the proof of recovering security can be expressed as an adaptation of their result; using the sponge as an extractor and the security of the next function. To formalise this:

Lemma 3.4.13.
Let $\mathsf{q}_\pi, \bar{q}_\pi := \mathsf{q}_\pi + Q(\mathsf{q}_\mathcal{D}), r, s, c$ *be as in Section 3.2.2. Let* $\epsilon_{\mathsf{ext}}(\mathsf{q}_\pi, \mathsf{q}_\mathcal{D})$ *be as described in [33, Section 5.3] and let* $\epsilon_{\mathsf{next}}(\bar{q}_\pi)$ *be the bound as in Lemma 3.4.2 as a function of* $\bar{q}_\pi$*; both with* n, r, c *as previously described. Given* Reverie*, also as in Section 3.2.2,* $\gamma^* > 0, \mathsf{q}_\mathcal{D} \geq 0, \mathcal{A}$*, a* q_π*-query adversary against recovering security, and* $\mathcal{D}$*, a* $(\mathsf{q}_\mathcal{D}, \mathsf{q}_\pi)$*-legitimate distribution sampler as defined in Definition 3.1.2. Then,*

$$\begin{aligned}\mathsf{Adv}^{(\gamma^*,\mathsf{q}_\pi)\text{-rec}}_{\mathsf{Rev}^\pi_{\mathsf{u,n,r}}}(\mathcal{A}, \mathcal{D}) &\leq \epsilon_{\mathsf{ext}}(\mathsf{q}_\pi + 1, \mathsf{q}_\mathcal{D}) + 2\epsilon_{\mathsf{next}}(\bar{q}_\pi) + \frac{\mathsf{q}_\pi}{2^{n-1}}\\ &\leq \frac{\bar{q}_\pi + 1}{2^{\gamma^*}} + \frac{Q(\mathsf{q}_\mathcal{D})}{2^{ur}} + \frac{7(\bar{q}_\pi^2 + 1) + 24\bar{q}_\pi}{2^{c-1}} + \frac{(\bar{q}_\pi + 1)d + d^2 + \mathsf{q}_\pi - 2\bar{q}_\pi}{2^{n-1}}.\end{aligned}$$

Proof outline: The strategy of the proof is to use the extractor properties of the sponge to replace the resulting state with a random state; following this the output of next will be random by the arguments of Lemma 3.4.2.

Proof. To formalise this, we require the construction of two adversaries, $\mathcal{A}_1, \mathcal{A}_2$ with the former being a $(\mathsf{q}_\pi + 1)$-adversary for the extraction lemma of [33] and the latter, a $\bar{q}_\pi$-adversary in the next distinguishing game. Then we have,

$$\mathsf{Adv}^{(\gamma^*,\mathsf{q}_\pi)-\mathsf{rec}}_{\mathsf{Rev}^\pi_{\mathsf{u,n,r}}}(\mathcal{A}, \mathcal{D}) \leq \mathsf{Adv}^{(\gamma^*,\mathsf{q}_\mathcal{D})\text{-ext}}_{\mathsf{Sp}_{\mathsf{n,r,u}}}(\mathcal{A}_1) + \mathsf{Adv}^{\mathsf{dist}}_{\mathcal{A}_2}(\mathsf{next}^\pi(\mathsf{U}_n), (\mathsf{U}_n, \mathsf{U}_r)). \quad (3.4.4)$$

Let $\mathcal{A}$ be the normal recovering security adversary, then $\mathcal{A}_1$ is built by running $\mathcal{A}$ on $\mathsf{seed}, \gamma_1, \ldots, \gamma_{\mathsf{q}_\mathcal{D}}, z_1, \ldots, z_{\mathsf{q}_\mathcal{D}}$ received from the challenger, $\mathcal{A}_1$ forwards any $\pi^\pm$ queries from $\mathcal{A}$ to the π oracle, along with any $\mathsf{get\text{-}refresh}$ oracle queries to the associated oracle. Once this has been done, $\mathcal{A}$ will output it's chosen pair (s_0, d), which $\mathcal{A}_1$ will again forward to the challenger as its chosen pair.

The challenger will then return the challenge s'_d and the remaining $\mathrm{I}_{k+d+1}, \ldots, \mathrm{I}_{\mathsf{q}_\mathcal{D}}$ to $\mathcal{A}_1$, which forwards the latter straight to $\mathcal{A}$ along with the output of $\mathsf{next}(s'_d)$ which it computes. $\mathcal{A}_1$ continues to forward any $\pi^\pm$ queries that $\mathcal{A}$ makes, before $\mathcal{A}$

makes it's guess b^*, which $\mathcal{A}_1$ forwards to the challenger as its own guess. Since $\mathcal{A}_1$ only forwards the queries $\mathcal{A}$ makes to $\pi^{\pm}$ together with calling $\mathsf{next}(s'_d)$, the query complexity of $\mathcal{A}_1$ is $\mathsf{q}_\pi + 1$.

For $\mathsf{b} = 0$, this simulates precisely the recovering security game, while $\mathsf{b} = 1$ corresponds to $\mathcal{A}$ receiving $(s, t) \leftarrow \mathsf{next}(\mathsf{U}_n)$, as opposed to the correct challenge $(s, t) \overset{\$}{\leftarrow} (\mathsf{U}_n, \mathsf{U}_r)$. This is considered in the second term of Equation (3.4.4). $\mathcal{A}_2$ is now constructed by simulating the $b = 1$ version of the extraction game, while running $\mathcal{A}_1$ and using the distinguishing challenge.

Finally, all that is left is to upper bound these advantages; [33, Lemma 6] yields

$$\mathsf{Adv}^{(\gamma^*,\mathsf{q}_\mathcal{D})\text{-ext}}_{\mathsf{Sp}_{\mathsf{n,r,u}}}(\mathcal{A}_1) \leq \epsilon_{\mathsf{ext}}(\mathsf{q}_\pi + 1, \mathsf{q}_\mathcal{D}) + \mathsf{Adv}^{(\gamma^*,\mathsf{q}_\mathcal{D})-\mathsf{hit}}_{\mathcal{D},\mathsf{n}}(\mathcal{A}_1),$$

where the latter value is precisely the probability that $\mathcal{A}_1$ queries $\pi^{-1}(s_d)$ in the ideal case. Since $\mathcal{A}_1$ is only making queries to π that $\mathcal{A}$ makes, this is in fact the probability that $\mathcal{A}$ queries $\pi^{-1}(s_d)$, and since $\mathcal{A}$ would either have to guess this value with probability $\frac{\mathsf{q}_\pi}{2^{n-1}}$ or have to invert the next challenge to have made this query, this is in fact the advantage of $\mathcal{A}_2$ playing the distinguishing game on the next function, albeit with $\bar{q}_\pi$ queries, due to the queries by the distribution sampler.

Thus, by Lemma 3.4.2 we have

$$\epsilon_{\mathsf{PRG}}(\bar{q}_\pi) \leq \left(5 - \frac{1}{2^r}\right)\frac{\bar{q}_\pi}{2^{c-1}},$$

and

$$\epsilon_{\mathsf{ext}}(\mathsf{q}_\pi + 1) \leq \frac{\bar{q}_\pi}{2^{\gamma^*}} + \frac{Q(\mathsf{q}_\mathcal{D})}{2^{ur}} + \frac{7(\bar{q}_\pi^2 + 2\bar{q}_\pi + 1)}{2^c} + \frac{(\bar{q}_\pi + 1)\mathsf{q}_\mathcal{D} + {\mathsf{q}_\mathcal{D}}^2}{2^{n-1}},$$

which completes the proof. □

Comparing our bound to the bound proved in [33], the expected reduction from $t+1$ to 1 calls is easily seen in the numerators, especially the $\frac{1}{2^n}$ term, which contains a large number of multiplied terms. Both bounds are still dominated by the extraction bound, though our bound overall is improved, especially in the situation with a small outer state size r, which is where the $\mathsf{sponge.prng}$ performs poorest due to the $\mathsf{p.forget}$ procedure.

3.5 Practical Comparison

For completeness, we ran timing tests of both Reverie and the sponge.prng with various parameters set, both utilising the relevant Keccak permutation. The implementations were done in python and included calls to the respective forward security measures of each design after each block of output. In keeping with the underlying Keccak permutations, we tested the constructions on $b = 400, 800$, and 1600, with $\ell = 512, 1024$, and 4096 respectively. For ease and time saving, we precomputed initial states for each test and set them for better analysis of the output mechanisms; this equates to a generator that has been initialised and absorbed sufficient entropy to be in a random state.

In Section 3.2.1, we theorised that, in terms of calls to the underlying permutation π, Reverie takes time $\left\lceil \frac{\ell}{r} \right\rceil$, while sponge.prng takes $1 + \left\lceil \frac{\ell}{r} \right\rceil \left\lceil \frac{c}{r} \right\rceil$ time. In reality, Reverie also has the more complex XOR while sponge.prng zeroes states but this should not affect findings too drastically.

3.5.1 Results

We present our predictions and findings in Table 3.1 with associated graphs (Figures 3.10a to 3.10c) for visual aid.

$b = 400, \ell = 512$

	Reverie	sponge.prng
Prediction	2π	$3\pi \approx 1.5\times$
$r = 256$	2.19ms	4.27ms
Actual	$1\times$	$\approx 2\times$
Prediction	4π	$9\pi \approx 2\times$
$r = 144$	3.27ms	10.4ms
Actual	$1\times$	$\approx 3\times$

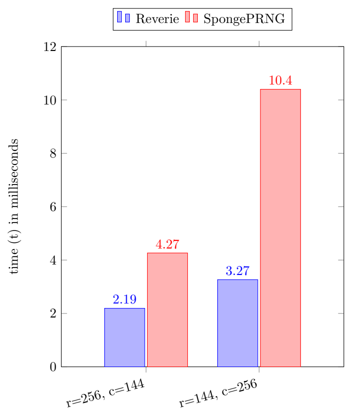
$b = 800, \ell = 1024$

	Reverie	sponge.prng
Prediction	2π	$3\pi \approx 1.5\times$
$r = 544$	1.3ms	3.41ms
Actual	$1\times$	$\approx 2.5\times$
Prediction	4π	$9\pi \approx 2\times$
$r = 256$	4.89ms	10.83ms
Actual	$1\times$	$\approx 2\times$

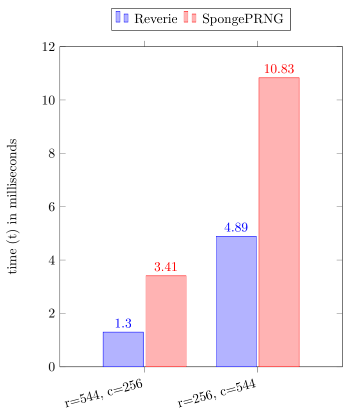
$b = 1600, \ell = 4096$

	Reverie	sponge.prng
Prediction	4π	$9\pi \approx 2\times$
$r = 1088$	4.64ms	9.5ms
Actual	$1\times$	$\approx 2\times$
Prediction	8π	$17\pi \approx 2\times$
$r = 576$	10.13ms	28.45ms
Actual	$1\times$	$\approx 3\times$

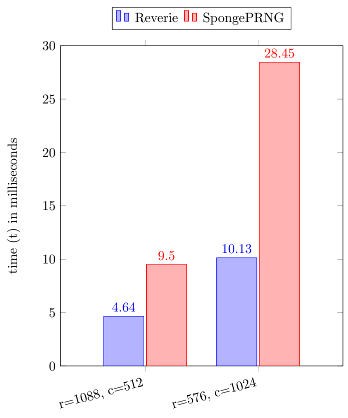
Table 3.1: Timing test results, where results given in terms of π are calls to the permutation π.



(a) Average timings of 1,000,000 iterations with $b = 400, \ell = 512$.



(b) Average timings of 1,000,000 iterations with $b = 800, \ell = 1024$.



(c) Average timings of 1,000,000 iterations with $b = 1600, \ell = 4096$.

3.5.2 Conclusion

Our results agree with our predictions, bar a few cases such as $b = 800.r = 544$ where Reverie performs better than expected. This could be due to several factors in the Python deployment, or the XOR vs zeroing steps in the constructions, but remains close to our predictions. Ideally, we would compare our results with well known Crypto libraries, but this would require implementing the generators in C or C++ with a higher level of knowledge of optimising software than the authors possess.

3.6 Extension to Parazoa

Although Reverie cannot be described as a sponge, it does fit into the generalised family of parazoa, as described in Section 2.6. We remove the seed mechanism in favour of the vanilla sponge padding scheme, together with the vanilla sponge's finalise algorithm that concatenates and truncates output. Removing the seed is necessary for both the padding and finalise functions to meet the requirements set out in Definitions 2.6.5 and 2.6.6. We also note that the reason for the seed is to "blind" adversarial input, whereas in the context of a parazoa function this is not necessary or could be incorporated into the f_1 function, with each seed hard coded into f_1. Following Definition 2.6.7, we describe Reverie as a $(\ell_b, r, \ell_{\mathsf{total}}, r, n)$-parazoa with f_1, f_2, g_1 as described in the sponge function, together with g_2 given as the feed forward operation.

Proposition 3.6.1.
For Reverie as defined above, the capacity loss $d = 0$.

Proof. This follows in the same way as the sponge construction. To be precise, the first criterion in Definition 2.6.8, the capacity loss d is 0. We require that for a fixed x and a fixed output r, $f_1(s, \mathrm{I}) = x$ for some I and $g_1(s) = r$. The former fixes the inner state which is left fixed by f_1 and is thus entirely defined by the value x, while the latter fixes the outer r-bits of s that match r. □

What remains is to prove that g_2 is a bijection on the state which given full knowledge of π, along with s_i, s_{i+1} is trivial.

Then by [3, Theorem 1] Reverie has

$$\mathsf{Adv}^{\mathsf{pro}}_{\mathsf{Rev}^{\pi},\mathsf{sim}}(\mathsf{D}) = O\left(\frac{((K+\ell_b)q_1 + q_2)^2}{2^{n-r}}\right).$$

This matches the derived indifferentiability bound of the sponge, meaning our design does not reduce security in this setting and may present an alternate avenue to prove robustness via indifferentiability from an idealised PRNG.

3.7 Conclusion

We have presented an updated construction, Reverie, for a sponge-like PRNG. The construction incorporates an effective and efficient forward-security mechanism and we have provided proofs of both preserving and recovering security in the chosen security model. Our design makes a single call to the permutation on every invocation of Reverie.next, while the comparable generators make $1 + t$ calls. Our design choice ensures the underlying permutation is called far fewer times. Thus, the loss of security from collisions is reduced when compared to the relevant bounds of other designs.

The main limiting factor of the bound relates to the recovering security bound; and more precisely the extraction bound. This begs the question: can this bound be improved? This is briefly discussed in [33] in the present setting, but we would also like to consider other, possibly similar, mechanisms that may present a better security bound; for instance, would a full state refresh yield a better bound? Work by Mennink, Reyhanitabar and Vizár [44] suggests there may be room for improvement. A full state refresh, however, enables in practise an adversary to more easily affect or even set the state of the generator. The generation of output could be modified to only make use of the feed-forward after several outputs to reduce the complexity of the next function. This would give rise to a second version of Reverie when viewed as a parazoa.

Chapter 4

Updated Security Model for PRNGs

Contents

This chapter extends the security model of Dodis et al.[29], motivated by the design of the NIST PRNGs [7]. The principle aim of this extension is to better capture the possible ways the NIST PRNGs may be used in practise, allowing for a generate subroutine that performs a "small" state update between outputs. Other aims include better modelling the setup phase of a PWI, allowing for more accurate analysis of cold boot situations where sufficient entropy may not be available to the generator. We begin by recalling and building upon the definition of a PWI (PRNG with input), with our own definition of a variable-output PWI. We then update the notion of a masking function from [55] with the idea of a split masking function, which will be useful in later security proofs when a state contains, by design, non-random parts. We then update the notion of robustness, followed by preserving and recovering security. Once these notions have been established, we are able to update the combination proof that implies robustness from having both preserving and recovering security.

4.1 Preliminaries

To analyse the NIST PRNG constructions we will require an extended definition of a PWI, building on [29] and the additions of non-random parts of the state with the use of masking functions by Shrimpton-Terashima [55]. Our extended definitions will allow calls to the next function to request a varying amount of output, within a limit, as opposed to a set amount. This will capture PWIs that have a subroutine and an extra input to deal with requests that are over a certain length, yet only update the internal state once. This ability for the adversary to vary the amount of output is an almost mirror of the update the authors of [29] made to the security notion of refreshing the generator from [4] to enable an adversary to slowly feed entropy into the generator as opposed to all at once. Shrimpton and Terashima [55] add another field called IFace which refers to the interface to which the adversary is making a request. However, this does not help us in terms of analysing the NIST constructions, unless an implementation allowed for an interface that called the PWI with the derivation function enabled and another without the derivation function. Consequently we only consider a single interface.

Recall the original definition of the distribution sampler as given in [29] and Definition 2.3.1.

Definition 4.1.1. The distribution sampler $\mathcal{D}$ is a stateful and probabilistic algorithm which, given the current state σ, outputs a tuple $(\sigma', \mathrm{I}, \gamma, z)$ where:

- σ' is the new state for $\mathcal{D}$,
- $\mathrm{I} \in \{0,1\}^p$ is the next input for the refresh algorithm,
- γ is some fresh entropy estimation of I,
- z is the leakage about I given to the adversary $\mathcal{A}$.

Let $\mathsf{q}_{\mathcal{D}}$ be the maximum number of calls to $\mathcal{D}$ in our security games. Then it is said that $\mathcal{D}$ is legitimate if, for all $j \in \{1, \ldots, \mathsf{q}_{\mathcal{D}}\}$,

$$H_\infty(\mathrm{I}_j | \mathrm{I}_1, \ldots, \mathrm{I}_{j-1}, \mathrm{I}_{j+1}, \ldots, \mathrm{I}_{\mathsf{q}_{\mathcal{D}}}, z_1, \ldots, z_{\mathsf{q}_{\mathcal{D}}}, \gamma_1, \ldots, \gamma_{\mathsf{q}_{\mathcal{D}}}) \geq \gamma_j,$$

where H_∞ is the minimum entropy function as defined in Definition 2.1.4.

We model an adversary using a pair $(\mathcal{A}, \mathcal{D})$, where $\mathcal{A}$ is the actual adversary and $\mathcal{D}$ is a stateful distribution sampler. The adversary $\mathcal{A}$'s goal is to determine a challenge bit b picked during the initialise procedure, which also returns the public parameters to the attacker.

The following definition is the updated definition from [55].

Definition 4.1.2 (PWI)**.** Let $p, \ell \in \mathbb{N}$, let IFace, $\mathsf{Seedspace}, \mathsf{Statespace}$, be non-empty sets. A PRNG with input (PWI) with interface set IFace, seed space $\mathsf{Seedspace}$, and state space $\mathsf{Statespace}$, is a tuple of deterministic algorithms
$G = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next}, \mathsf{tick})$, where

- setup: takes no input, and generates an initial PWI state $s_0 \in \mathsf{Statespace}$. Although setup itself is deterministic, it may be provided oracle access to an entropy source $\mathcal{D}$, in which case its output s_0 will be a random variable determined by the random coins of $\mathcal{D}$.
- refresh: $\mathsf{Seedspace} \times \mathsf{Statespace} \times \{0,1\}^p \longrightarrow \mathsf{Statespace}$ is a deterministic algorithm that takes a seed $\mathsf{seed} \in \mathsf{Seedspace}$, the current PWI state $s_i \in \mathsf{Statespace}$, and a string $\mathrm{I}_j \in \{0,1\}^p$ as input, and returns a new state $s_{i+1} \in \mathsf{Statespace}$.
- next: $\mathsf{Seedspace} \times \mathsf{IFace} \times \mathsf{Statespace} \longrightarrow \mathsf{Statespace} \times (\{0,1\}^\ell \cup \{\perp\})$ is a deterministic algorithm that, given seed, an interface label $m \in \mathsf{IFace}$, and the current state $s_i \in \mathsf{Statespace}$, returns a new state $s_{i+1} \in \mathsf{Statespace}$, and either an ℓ-bit output value $r_{i+1} \in \{0,1\}^\ell$ or a distinguished symbol $\perp$.
- tick: $\mathsf{Seedspace} \times \mathsf{Statespace} \longrightarrow \mathsf{Statespace}$ is a deterministic algorithm that takes the seed $\mathsf{seed} \in \mathsf{Seedspace}$ and the current state s_i as input, and returns a new state s_{i+1}.

It should be noted that this definition assumes that the seed is generated externally and provided to the PWI. The definition of a PWI is illustrated in Figure 4.1.

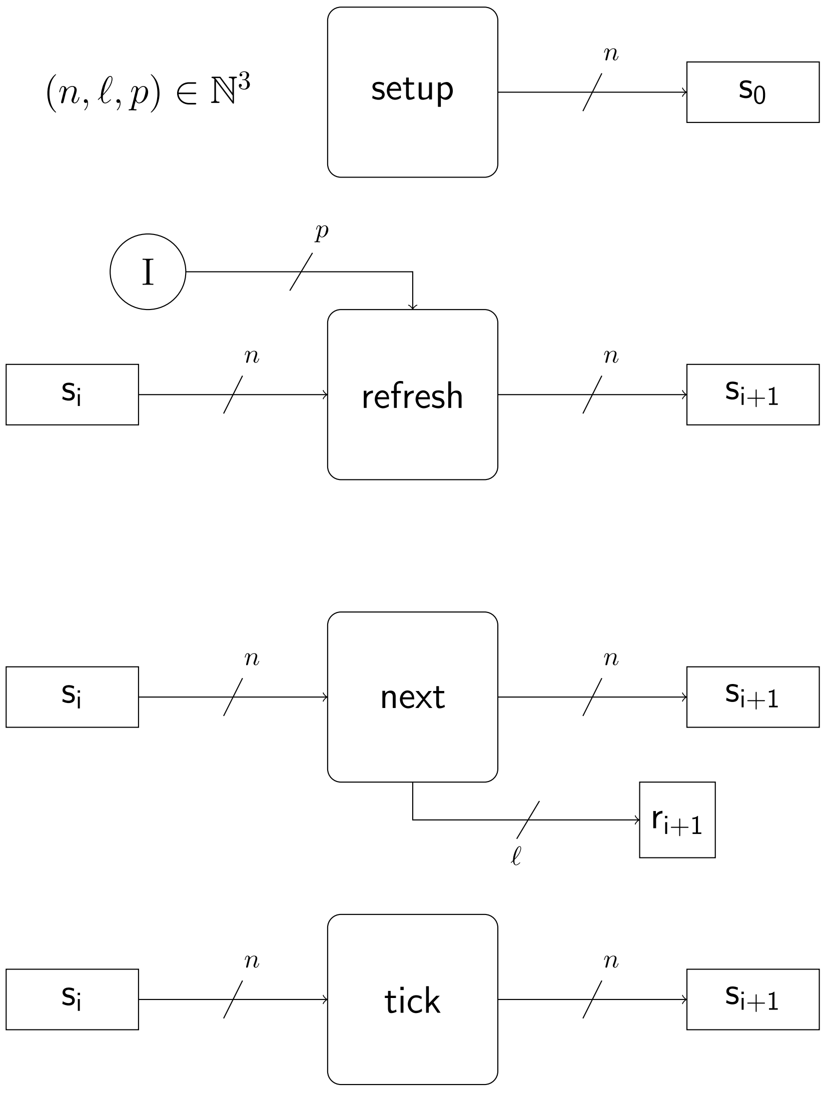


Figure 4.1: Definition of a PWI.

We extend Definitions 4.1.1 and 4.1.2 in several ways:

- The distribution sampler will output an initial value I_0 which may be a concatenation of several outputs, which is independent of all other outputs and is used solely in the setup algorithm to generate the initial generator state.
- We add functionality to next so that each call can request a specific amount of output, bounded above by a parameter ℓ_{max}. This reflects the capability of some generators to generate a varying amount of output per state update, often using a different subroutine to do so, or by just truncating output.
- We add a separate algorithm seedgen to model the generation of the seed. This is to capture the notion that in practice the seed cannot in general be chosen uniformly at random; the seed must be generated using either system entropy or be a fixed value. In our theoretical setting however, this becomes restrictive on $\mathcal{D}$ since there must be a certain amount of separation between entropy used to generate the seed and entropy inputs to the generator.

  One alternative option would be to have an entirely separate entropy source just for generating the seed. This is unlikely to be the case in practice due to the difficultly in providing good, independent entropy sources for the generator. There is also the question of whether providing more "good" entropy sources would negate the need for the seed; even if each entropy source was again modelled adversarially, albeit without the ability to communicate with the distinguisher or other entropy sources.
- We modify the algorithm setup slightly to take as input the public seed seed, which may be used in creation of the initial state s_0. The algorithm setup is also given access to the entropy source. The latter generates a special I_0 that contributes to the creation of the initial state. We add several changes to $\mathcal{D}$ to reflect this change and how this entropic I_0 must be independent of the other entropy values produced by $\mathcal{D}$.

  The changes made to the setup algorithm are a conscious choice that could easily have been incorporated into the refresh algorithm. For example, a PWI state could include a single bit that represents whether the generator has been initialised. If this bit is set to 0 then the refresh algorithm would run in a different way to normal that would represent the initialisation of the state of the PWI. We decided to alter the definition of the setup algorithm to be more concrete in separating and highlighting this aspect of the design of a PWI.

4.2 Definition of a VOPWI

Formally, we have the following definition of a Variable-Output PRNG with Input:

Definition 4.2.1 (VOPWI)**.** Let $p, \ell_i, \ell_{\mathsf{max}} \in \mathbb{N}$, let $\mathsf{IFace}, \mathsf{Seedspace}, \mathsf{Statespace}$, be non-empty sets. A variable-output PRNG with input (VOPWI) with maximum output size $\ell_{\mathsf{max}} \geq \ell_i$, with interface set IFace, seed space Seedspace, and state space Statespace, is a tuple of deterministic algorithms $G = (\mathsf{seedgen}, \mathsf{setup}, \mathsf{refresh}, \mathsf{next}, \mathsf{tick})$, where

- seedgen is a randomised algorithm that outputs $\mathsf{seed} \in \mathsf{Seedspace}$.
- setup is a deterministic algorithm that takes input seed, and is provided access to the entropy source $\mathcal{D}$ which passes I_0 to setup. setup generates and outputs an initial VOPWI state $s_0 \in \mathsf{Statespace}$.
- refresh: $\mathsf{Seedspace} \times \mathsf{Statespace} \times \{0,1\}^p \longrightarrow \mathsf{Statespace}$ is a deterministic algorithm that takes a seed $\mathsf{seed} \in \mathsf{Seedspace}$, the current PWI state $s_i \in \mathsf{Statespace}$, and a string $\mathrm{I}_j \in \{0,1\}^p$ as input, and returns a new state $s_{i+1} \in \mathsf{Statespace}$.
- next: $\mathsf{Seedspace} \times \mathsf{IFace} \times \mathsf{Statespace} \times 1^{\leq \ell_{\mathsf{max}}} \rightarrow \mathsf{Statespace} \times (\{0,1\}^{\leq \ell_{\mathsf{max}}} \cup \{\perp\})$ is a deterministic algorithm that, given seed, an interface label $m \in \mathsf{IFace}$, the current state s_i, and the encoding of an integer $\ell_{i+1} \leq \ell_{\mathsf{max}}$, and returns a new state s_{i+1}, and either an output value $r_{i+1} \in \{0,1\}^{\ell_{i+1}}$ or a distinguished, symbol $\perp$. If the generator never outputs $\perp$ then it is called "non-blocking".
- tick: $\mathsf{Seedspace} \times \mathsf{Statespace} \longrightarrow \mathsf{Statespace}$ is a deterministic algorithm that takes a seed seed and the current state $s_i \in \mathsf{Statespace}$ as input, and returns a new state $s_{i+1} \in \mathsf{Statespace}$.

The definition of a VOPWI is illustrated in Figure 4.2.

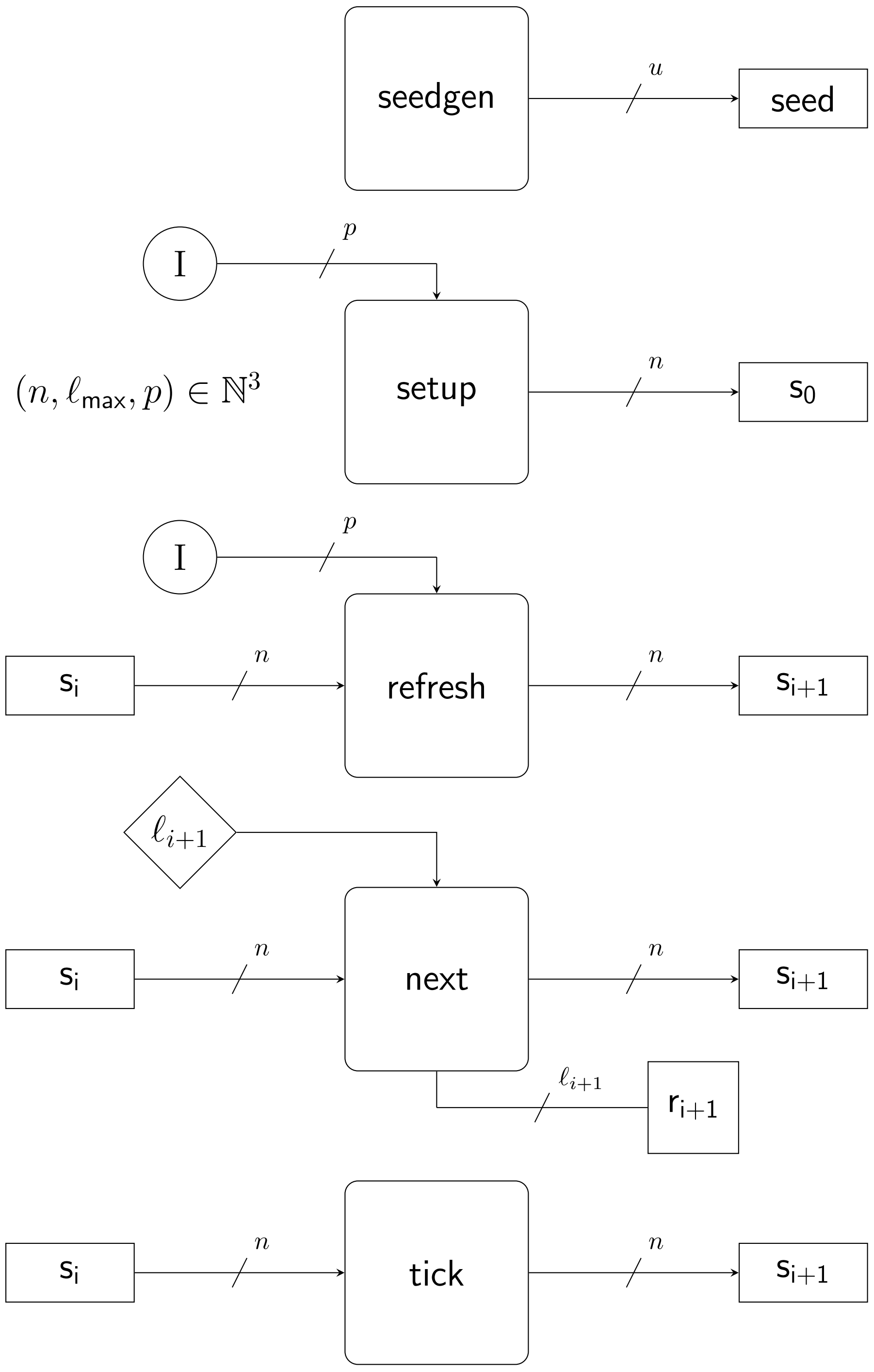


Figure 4.2: Definition of a VOPWI.

Definition 4.2.2. The distribution sampler $\mathcal{D}$ is a stateful and probabilistic algorithm which, given the current state σ, outputs a tuple $(\sigma', \mathrm{I}, \gamma, z)$ where:

- σ' is the new state for $\mathcal{D}$,
- $\mathrm{I} \in \{0,1\}^p$ is the next input for the refresh algorithm,
- γ is some fresh entropy estimation of I,
- z is the leakage about I given to the adversary $\mathcal{A}$.

In addition, we require that the distribution sampler outputs a special initial value I_0, which we define as the totality of outputs from $\mathcal{D}$ given as input to the algorithm setup. We do the same for γ_0 and z_0. Often, we will assume γ_0 is enough to ensure the initial state generated is uniformly random, and $z_0 = \emptyset$. We will state this explicitly when we assume it.

Let $\mathsf{q}_{\mathcal{D}}$ be the maximum number of calls to $\mathcal{D}$ in our security games. Then it is said that $\mathcal{D}$ is legitimate if, for all $j \in \{0, \ldots, \mathsf{q}_{\mathcal{D}}\}$,

$$H_\infty(\mathrm{I}_j | \mathrm{I}_0, \ldots, \mathrm{I}_{j-1}, \mathrm{I}_{j+1}, \ldots, \mathrm{I}_{\mathsf{q}_{\mathcal{D}}}, z_0, \ldots, z_{\mathsf{q}_{\mathcal{D}}}, \gamma_0, \ldots, \gamma_{\mathsf{q}_{\mathcal{D}}}) \geq \gamma_j.$$

Because of the special requirements on the initial I_0 we now also require I_0 to be independent of all the other refresh values, leakage and estimates and to satisfy a new minimum entropy requirement γ_0.

It may be possible that we only require

$$H_\infty(\mathrm{I}_0 | \mathrm{I}_1, \ldots, \mathrm{I}_{\mathsf{q}_{\mathcal{D}}}, z_0, \ldots, z_{\mathsf{q}_{\mathcal{D}}}, \gamma_0, \ldots, \gamma_{\mathsf{q}_{\mathcal{D}}}) = H_\infty(\mathrm{I}_0) \geq \gamma_0.$$

This is a slight weakening of independence since there may be a difference in the actual amount of entropy of I_0. However, as an example, we could define a distribution sampler such that the first bit of I_0 and I_1 are always the same and skewed towards 1, while the remaining bits are chosen uniformly at random. The minimum entropy of I_0 given I_1 would be equal, but this first bit could dictate a particular way the setup algorithm is run. This knowledge gives an adversary in possession of the first bit of I_1 a non-negligible advantage.

4.3 Masking Functions

In this section we will provide an overview of masking functions, as introduced in [55]. before updating them for our purposes and our new definition of a VOPWI.

Masking functions allow for non-uniform state, be it a counter or other unchanging or predictively changing parts of the state. It is still required that a part of the state is unpredictable to an attacker. This allows for proper modelling of generators that use states consisting of non-random parts along with an unpredictable, high-entropy part.

Formally, a masking function is defined as follows:

Definition 4.3.1 (Masking Function)**.** Let $\mathsf{Statespace}$ be as defined in Definition 4.1.2. A masking function M is a randomised algorithm $\mathsf{M} : \mathsf{Statespace} \to \mathsf{Statespace}$, that takes a state as input and outputs an ideal state.

As an example, let a state be of the form (a, b), where a is a collection of static or predictable fields, while b is a high-entropy buffer, and define $\mathsf{M}(s) := (a, b')$ where b' is sampled from some distribution by M.

The purpose of a masked state is to capture what characterises a "good" or perfect PWI state, for example, $\mathsf{M}(s)$ should be indistinguishable from a state s that has accumulated enough entropy. Since a masked state should produce such a "good" state, a PWI called on such a state should produce pseudo-random output.

Of particular interest is the initial state of the PWI, s_0. $\mathsf{Initialise}$ is modified slightly from the original definition given in Figure 2.4 as described in Figure 4.3; where the setup algorithm is given access to the entropy source. At this point we start thinking of an entropy source $\mathcal{D}$, which we will later place restrictions on similar to the distribution sampler. The behaviour of $\mathsf{Initialise}$ motivates the following definition:

Definition 4.3.2 (Honest-Initialisation Masking Functions)**.** Let $\mathcal{D}$ be an entropy source, $\mathsf{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next}, \mathsf{tick})$ be a PWI with statespace $\mathsf{Statespace}, \mathcal{A}$ an adversary and $\mathsf{M} : \mathsf{Statespace} \to \mathsf{Statespace}$ a masking function. Let (seed, Z) be a random variable output by running the initialise procedure (Figure 4.3), and s_0 be

```
Procedure: Initialise ()
$\sigma \leftarrow 0, e \leftarrow 0, x \leftarrow -1$
$\mathsf{seed} \xleftarrow{\$} \mathsf{seedgen}$
$\boxed{\mathrm{I}_0 \xleftarrow{\$} \mathsf{ES}}$
$s_0 \leftarrow \boxed{\mathsf{setup}(\mathrm{I}_0)}$
$\boxed{e \leftarrow \gamma_0}$
$\boxed{\textbf{if } e \geq \gamma^* \textbf{ then}}$
    $\mathsf{corrupt} \leftarrow \mathsf{false}$
$\mathsf{b} \xleftarrow{\$} \{0,1\}$
$\textbf{return } (\mathsf{seed}, (\gamma_0, z_0))$
```

Figure 4.3: The updated procedure Initialise. The boxed item denotes the change from the original definition in Figure 2.4.

the PWI state produced by this procedure. Set

$$\mathsf{Adv}^{\mathsf{init}}_{\mathsf{G},\mathcal{D},\mathsf{M}}(\mathcal{A}) = \Pr\left[\mathcal{A}(s_0, \mathsf{seed}, Z) \implies 1\right] - \Pr\left[\mathcal{A}(\mathsf{M}(s_0), \mathsf{seed}, Z) \implies 1\right].$$

If $\mathsf{Adv}^{\mathsf{init}}_{\mathsf{G},\mathcal{D},\mathsf{M}}(\mathcal{A}) \leq \epsilon_{\mathsf{h}}$ for all adversaries $\mathcal{A}$ running in time t, then M is said to be a $(\mathsf{G}, \mathcal{D}, t, \epsilon_{\mathsf{h}})$-honest-initialisation masking function.

The definition of an honest-initialisation masking function is made with respect to a specific entropy source $\mathcal{D}$, making the assumptions required of $\mathcal{D}$ dependent on the PWI in question, but these assumptions should be as weak as possible.

Following this definition we now define the notion of "bootstrapped" security, which refers to when a PWI starts from an "ideal" state, i.e. what we expect after a secure initialisation of the system.

Definition 4.3.3 (Bootstrapped Security)**.** Let G be a PWI and M be a masking function. For $x \in \{\mathsf{fwd}, \mathsf{bwd}, \mathsf{res}, \mathsf{rob}\}$ as defined in Section 2.3.1, let $\mathsf{Adv}^{\mathsf{x/M}}_{\mathsf{G},\mathcal{D}}(\mathcal{A})$ be defined as $\mathsf{Adv}^{\mathsf{x}}_{\mathsf{G},\mathcal{D}}(\mathcal{A})$ (as defined in Definition 2.3.6), but changing the procedure Initialise to the procedure given in Figure 4.4.

These new notions are useful, since they allow us to work with an idealised initial state. However, it is necessary to prove that the masking function fully and

```
Procedure: Initialise ()
$\sigma \leftarrow 0, e \leftarrow 0, x \leftarrow -1$
seed $\xleftarrow{\$}$ seedgen
$\mathrm{I}_0 \xleftarrow{\$}$ ES
$s_0 \leftarrow$ setup$(\mathrm{I}_0)$
$e \leftarrow \gamma_0$
$\boxed{s_0 \leftarrow \mathsf{M}(s_0)}$
if $e \geq \gamma^*$ then
    corrupt $\leftarrow$ false
b $\xleftarrow{\$} \{0,1\}$
return (seed, $(\gamma_0, z_0)$)
```

Figure 4.4: The updated procedure Initialise for bootstrapped security. The boxed item denotes the change from the definition in Figure 4.3.

accurately reflects the setup procedure being run.

Theorem 4.3.4 (from Theorem 1.6 of [55]).
Let G *be a* PWI*,* $\mathcal{D}$ *an entropy source, and* M *a masking function. Suppose that* M *is a* $(\mathsf{G}, \mathcal{D}, t, \epsilon_{\mathsf{h}})$*-honest initialisation mask. Then for any* $x \in \{\mathsf{fwd}, \mathsf{bwd}, \mathsf{res}, \mathsf{rob}\}$ *as defined in Section 2.3.1, there exists some adversary* $\mathcal{B}(\cdot)$ *such that for any adversary* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathsf{x}}_{\mathsf{G},\mathcal{D}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{x}/\mathsf{M}}_{\mathsf{G},\mathcal{D}}(\mathcal{B}(\mathcal{A})) + \epsilon_{\mathsf{h}}.$$

If it takes time t' *to compute* M*,* $\mathcal{A}$ *makes* q *queries and runs in time* t*, then* $\mathcal{B}(\mathcal{A})$ *makes* q *queries and runs in time* $\mathcal{O}(t) + t'$*.*

The authors of [55] proceed by proving an analogy of [29, Theorem 1]. The theorem shows that if a PWI satisfies two simpler notions of security, called preserving and recovering security, as defined in Definitions 2.3.7 and 2.3.8, then the PWI in question satisfies the conditions of robustness in the Shrimpton-Terashima setting. This Shrimpton-Terashima setting refers to the authors' extended definitions of the original definitions of [29].

Theorem 4.3.5 (Informal)**.**
Let G be a PWI and suppose there exists a split masking function M such that

1. *When starting from an arbitrary initial state s_0 of the adversary's choosing, the final PWI state s is indistinguishable from $\mathsf{M}(s_0')$, provided the PWI obtains sufficient entropy specified by the construction. This requirement is formalised as recovering security.*
2. *When starting from an initial state $\mathsf{M}(s_0')$ (adversarially chosen s_0'), the final PWI state s is indistinguishable from $\mathsf{M}(s)$, even if the adversary controls the intervening entropy inputs. This requirement is formalised as preserving security.*
3. *G produces pseudo-random outputs when in a masked state.*

Then G is robust.

The following additional property is required of the masking function for use in the proof of robustness and is as stated in [55, Definition 8].

Definition 4.3.6 (Idempotent Masking Functions)**.** A masking function $\mathsf{M} : \{0,1\}^n \to \{0,1\}^n$ is idempotent if, for any state $s \in \{0,1\}^n$, $\mathsf{M}(s)$ and $\mathsf{M}(\mathsf{M}(s))$ are identically distributed random variables.

4.4 Updated Security Notions

Having summarised the modifications of [55] to the definitions of [29], we extend their definitions for analysis of the NIST generators. We start by updating the existing security notions to encompass our changes in Figure 4.5.

Definition 4.4.1 (Variable-Output Robustness)**.** A Variable-Output PRNG with input is called $(t, \ell_{\mathsf{max}}, \ell_{\mathsf{total}}, \mathsf{q}_{\mathcal{D}}, \mathsf{q}_{\mathsf{S}}, \gamma^*, \epsilon_{\mathsf{rob}})$-robust $(\mathsf{rob}^{\ell_{\mathsf{total}}})$ if for any attacker $\mathcal{A}$

```
Procedure: Initialise ()
    $\sigma \leftarrow 0, e \leftarrow 0, x \leftarrow -1$
    $\mathsf{seed} \xleftarrow{\$} \mathsf{seedgen}$
    $\boxed{\mathrm{I}_0 \xleftarrow{\$} \mathsf{ES}}$
    $s_0 \leftarrow \boxed{\mathsf{setup}(\mathrm{I}_0)}$
    $\boxed{e \leftarrow \gamma_0}$
    $\boxed{\textbf{if } e \geq \gamma^* \textbf{ then}}$
        $\mathsf{corrupt} \leftarrow \mathsf{false}$
    $\mathsf{b} \xleftarrow{\$} \{0,1\}$
    return $(\mathsf{seed}, \boxed{(\gamma_0, z_0)})$

Procedure: Finalise ($\mathsf{b}^*$)
    if $\mathsf{b} = \mathsf{b}^*$ then
        return 1
    else
        return 0

Oracle: ES()
    $x \leftarrow x + 1$
    $(\sigma', \mathrm{I}_x, \gamma_x, z_x) \xleftarrow{\$} \mathcal{D}(\sigma)$
    return $\mathrm{I}_x$

Procedure: get-state ()
    $e \leftarrow 0$
    $\mathsf{corrupt} \leftarrow \mathsf{true}$
    return $s_i$

Procedure: set-state ($s^*$)
    $e \leftarrow 0$
    $\mathsf{corrupt} \leftarrow \mathsf{true}$
    $s_i \leftarrow s^*$

Procedure: next-ror ($\ell_{i+1}$)
    $(s_{i+1}, r_0) \leftarrow \boxed{\mathsf{next}(s_i, \ell_{i+1})}$
    if $\mathsf{corrupt} = \mathsf{true}$ then
        $e \leftarrow 0$
        return $r_0$
    else
    if $r_0 = \perp$ then
        $r_1 \leftarrow \perp$
    else
        $r_1 \xleftarrow{\$} \boxed{\{0,1\}^{\ell_{i+1}}}$
    return $r_{\mathsf{b}}$

Procedure: get-next ($\ell_{i+1}$)
    $(s_{i+1}, r_{i+1}) \leftarrow \boxed{\mathsf{next}(s_i, \ell_{i+1})}$
    if $\mathsf{corrupt} = \mathsf{true}$ then
        $e \leftarrow 0$
    return $r_{i+1}$

Procedure: wait()
    $s_{i+1} \leftarrow \mathsf{tick}(\mathsf{seed}, s_i)$

Procedure: $\mathcal{D}$−refresh()
    $x \leftarrow x + 1$
    $(\sigma, \mathrm{I}, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$
    $s_{i+1} \leftarrow \mathsf{refresh}(s_i, \mathrm{I})$
    $e \leftarrow e + \gamma$
    if $e \geq \gamma^*$ then
        $\mathsf{corrupt} \leftarrow \mathsf{false}$
    return $(\gamma, z)$
```

Figure 4.5: The updated security procedures, updated from the original definitions given in [29, 55]. Boxed items indicate changes. The inclusion of seed has been omitted from several algorithms.

running in time at most t, making at most $\mathsf{q}_{\mathcal{D}}$ calls to $\mathcal{D}-\mathsf{refresh}$, requesting at most ℓ_{total} bits of cumulative output from the VOPWI, and no more than ℓ_{max} from each next call, q_{S} calls to $\mathsf{get\text{-}state/set\text{-}state}$, and any legitimate distribution sampler $\mathcal{D}$, the advantage in the game specified in Figure 4.5 is at most ϵ. The value γ^* is the minimum entropy required to reset the "corrupt" flag back to "false". As usual, the challenger first executes the $\mathsf{Initialise}$ algorithm and the adversary is given access to the bottom six oracles. Once the adversary has asked all her queries, she outputs her guess, passes it to the challenger and the challenger runs $\mathsf{Finalise}$ with the adversary's guess as input.

Further, we define three more games which are restrictions of the robustness game:

Definition 4.4.2. Resilience ($\mathsf{res}^{\ell_{\mathsf{total}}}$) is the restricted game where $\mathsf{q}_{\mathsf{S}} = 0$.

Definition 4.4.3. Forward-secure ($\mathsf{fwd}^{\ell_{\mathsf{total}}}$) is the restricted game where $\mathcal{A}$ makes no calls to $\mathsf{set\text{-}state}$ and a single call to $\mathsf{get\text{-}state}$, which must be the very *last* oracle call that $\mathcal{A}$ makes.

Definition 4.4.4. Backward-secure ($\mathsf{bwd}^{\ell_{\mathsf{total}}}$) is the restricted game where $\mathcal{A}$ makes no calls to $\mathsf{get\text{-}state}$ and a single call to $\mathsf{set\text{-}state}$ which is the very *first* call $\mathcal{A}$ makes.

Variable-Output Robustness implies all three other notions and is the strongest notion. Resilience is the most restricted notion; it captures security against arbitrary distribution samplers when the VOPWI is not corrupted. Forward security protects past VOPWI outputs in the event that a state is compromised. Backward security protects future VOPWI outputs if enough entropy is input into the system. Since the NIST generators have states that update in very situational-dependent ways, we define a split masking function, specifically a masking function that behaves in different ways depending on the situation.

Definition 4.4.5 (Split Masking Functions)**.** Let $\mathsf{Statespace}$ be as defined in Definition 4.2.1, and define $\mathsf{M} := (\mathsf{M}_{\mathsf{I}}, \mathsf{M}_{\mathsf{P}}, \mathsf{M}_{\mathsf{R}})$ to be a tuple of randomised algorithms $\mathsf{M}_{\mathsf{I}}, \mathsf{M}_{\mathsf{P}}, \mathsf{M}_{\mathsf{R}} : \mathsf{Statespace} \to \mathsf{Statespace}$.

Each algorithm is used at a specific point in the security games. M_{I} is used in $\mathsf{Initialise}$, M_{P} is used after a preserving call to next, and lastly, M_{R} is used after a recovering call to next. We call M a split masking function over $\mathsf{Statespace}$. By using a split masking function, we allow for different behaviours during the running of the VOPWI, which correspond to how the notion of "ideal state" changes depending on the situation. For example, after recovering from insufficient entropy, we might expect the state to look random over several fields, while on the other hand, on a preserving next call, where there is no additional entropy available we might expect some of these fields to remain constant.

As an example, for a state $s := (a, b, c)$, where $a, b \in \{0,1\}^n$ are "working states" and c is a counter, a masking function could be as follows:

$$\begin{aligned}
\mathsf{M_I}(s) &= (a', f(a'), 0x1) & a' &\xleftarrow{\$} \{0,1\}^n && \text{for some function } f\text{, during initialisation,}\\
\mathsf{M_P}(s) &= (a', b, c) & a' &\xleftarrow{\$} \{0,1\}^n && \text{after a preserving next call,}\\
\mathsf{M_R}(s) &= (a', b', c) & a', b' &\xleftarrow{\$} \{0,1\}^n && \text{after a recovering next call.}
\end{aligned}$$

The example reads as follows:

- After initialisation the first field is randomly sampled, the second field is some function of the first field, while the last field, the counter, is initialised.
- After a preserving next call the first field is again sampled from random, while the second and third fields remain constant.
- Lastly, after a recovering next call, the first and second fields are sampled from random, while the third field remains constant.

Definition 4.4.6 (Idempotent Split Masking Functions)**.** A split masking function $\mathsf{M} = (\mathsf{M_I}, \mathsf{M_P}, \mathsf{M_R}) : \{0,1\}^n \to \{0,1\}^n$ is idempotent if, for any state $s \in \{0,1\}^n$, and for any $\mathsf{a} \in \{\mathsf{P}, \mathsf{R}\}, \mathsf{b} \in \{\mathsf{I}, \mathsf{P}, \mathsf{R}\}$, $\mathsf{M_a}(s)$ and $\mathsf{M_a}(\mathsf{M_b}(s))$ are identically distributed random variables.

We note that Definition 4.4.6 is slightly stronger than necessary for the purpose of the robustness proof which follows in Section 4.5.

To complete the updated notions we also need to modify Definitions 4.3.2 and 4.3.3, together with Theorem 4.3.4. We present these in Definitions 4.4.7 and 4.4.8.

```
Procedure: Initialise ()
$\sigma \leftarrow 0, e \leftarrow 0, x \leftarrow -1$
$\mathsf{seed} \xleftarrow{\$} \mathsf{seedgen}$
$s_0 \leftarrow \mathsf{setup}^{\mathsf{ES}}$
$e \leftarrow \gamma_0$
$\boxed{s_0 \leftarrow \mathsf{M_I}(s_0)}$
if $e \geq \gamma^*$ then
    $\mathsf{corrupt} \leftarrow \mathsf{false}$
$\mathsf{b} \xleftarrow{\$} \{0,1\}$
return $(\mathsf{seed}, (\gamma_0, z_0))$
```

Figure 4.6: The updated procedure Initialise for Variable-Output bootstrapped security. Updated from Figure 4.4.

Definition 4.4.7 (Honest-Initialisation Split Masking Functions)**.** Let $\mathcal{D}$ be an entropy source, $\mathsf{G} = (\mathsf{seedgen}, \mathsf{setup}, \mathsf{refresh}, \mathsf{next}, \mathsf{tick})$ be a VOPWI with statespace Statespace, $\mathcal{A}$ an adversary and $\mathsf{M} : \mathsf{Statespace} \to \mathsf{Statespace}$ a split masking function. Let (seed, Z) be a random variable output by running the Initialise procedure (Figure 4.3), and s_0 be the VOPWI state produced by this procedure. Set

$$\mathsf{Adv}^{\mathsf{init}}_{\mathsf{G},\mathcal{D},\mathsf{M}}(\mathcal{A}) = \Pr\left[\mathcal{A}(s_0, \mathsf{seed}, Z) \implies 1\right] - \Pr\left[\mathcal{A}(\mathsf{M_I}(s_0), \mathsf{seed}, Z) \implies 1\right].$$

If $\mathsf{Adv}^{\mathsf{init}}_{\mathsf{G},\mathcal{D},\mathsf{M}}(\mathcal{A}) \leq \epsilon$ for all adversaries $\mathcal{A}$ running in time t, then M is said to be a $(\mathsf{G}, \mathcal{D}, t, \epsilon_{\mathsf{h}})$-honest-initialisation split masking function.

Definition 4.4.8 (Bootstrapped Security)**.** Let $\mathsf{G}^{\ell_{\mathsf{max}}}$ be a VOPWI with maximum output ℓ_{max}-bits, and let M be a split masking function. For $x \in \{\mathsf{fwd}^{\ell_{\mathsf{total}}}, \mathsf{bwd}^{\ell_{\mathsf{total}}}, \mathsf{res}^{\ell_{\mathsf{total}}}, \mathsf{rob}^{\ell_{\mathsf{total}}}\}$ as defined in Definitions 4.4.1 to 4.4.4, let $\mathsf{Adv}^{\mathsf{x/M}}_{\mathsf{G}^{\ell_{\mathsf{max}}},\mathcal{D}}(\mathcal{A})$ be defined as $\mathsf{Adv}^{\mathsf{x}}_{\mathsf{G}^{\ell_{\mathsf{max}}},\mathcal{D}}(\mathcal{A})$, but changing the procedure Initialise to the procedure given in Figure 4.6.

Theorem 4.4.9.
Let $\mathsf{G}^{\ell_{max}}$ be a VOPWI with maximum output size ℓ_{max}, $\mathcal{D}$ an entropy source, and M a split masking function. Suppose that M is a $(\mathsf{G}, \mathcal{D}, t, \epsilon_{\mathsf{h}})$-honest initialisation split mask. Then for any $x \in \{\mathsf{fwd}^{\ell_{total}}, \mathsf{bwd}^{\ell_{total}}, \mathsf{res}^{\ell_{total}}, \mathsf{rob}^{\ell_{total}}\}$ as defined in Definitions 4.4.1 to 4.4.4, there exists some adversary $\mathcal{B}(\cdot)$ such that for any adversary $\mathcal{A}$,

$$\mathsf{Adv}^{\mathsf{x}}_{\mathsf{G}^{\ell_{max}},\mathcal{D}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{x/M}}_{\mathsf{G}^{\ell_{max}},\mathcal{D}}(\mathcal{B}(\mathcal{A})) + \epsilon_{\mathsf{h}}.$$

If it takes time t' to compute M*,* $\mathcal{A}$ *makes* q *queries and runs in time t, then* $\mathcal{B}(\mathcal{A})$ *makes* q *queries and runs in time* $\mathcal{O}(t) + t'$*.*

Proof. This follows directly from the proof of Theorem 4.3.4 since our split masking function acts precisely as the original masking function in Initialise. The only other change introduced that is relevant to the Initialise procedure is the addition of the special entropy input I_0 which is used by setup to create the initial VOPWI state.

The changes we have made only affect the output of the setup algorithm. This leaves the rest of the game untouched. □

4.5 Variable-Output Robustness

To obtain an analogy of the robustness theorem of [29, Theorem 1] we must first adapt the notions of preserving and recovering security to capture the updates that we made in Section 4.4.

Recall, preserving security concerns the situation where the VOPWI starting with a “good” state, remains “good” even after being refreshed with adversarially controlled inputs. Recovering security concerns the situation where the VOPWI has been compromised to an adversarially chosen state, but is then refreshed with sufficient entropy from multiple calls to $\mathcal{D}-$refresh such that the corrupt flag is set back to false, resulting in a “good” state. Both are measured by the adversary’s ability to distinguish the resulting state and generator output from an ideal state and random output.

In Figure 4.7 we present the updated preserving and recovering security games with the changes from the original definitions given in [29, Definitions 3 & 4] highlighted.

```
$\underline{\mathsf{Recover}(\mathsf{G}, \mathcal{A}, \mathcal{D}, \mathsf{M}, \ell)}$
$(\mathsf{seedgen}, \mathsf{setup}, \mathsf{refresh}, \mathsf{next}, \mathsf{tick}) \leftarrow \mathsf{G}$
$\mathsf{seed} \leftarrow \mathsf{seedgen}; \mathsf{b} \overset{\$}{\leftarrow} \{0,1\}$
$\sigma_0 \leftarrow 0; \mu \leftarrow 0$
**for** $k = 1, \ldots, \mathsf{q}_{\mathcal{D}}$ **do**
  $(\sigma_k, \mathrm{I}_k, \gamma_k, z_k) \leftarrow \mathcal{D}(\sigma_{k-1})$
$(s_0, d, \sigma') \overset{\$}{\leftarrow} \mathcal{A}^{\mathsf{get\text{-}refresh}}(\mathsf{seed},$
$\gamma_1, \ldots, \gamma_{\mathsf{q}_{\mathcal{D}}}, z_1, \ldots, z_{\mathsf{q}_{\mathcal{D}}})$
**if** $\mu + d > \mathsf{q}_{\mathcal{D}}$ or $\sum_{j=\mu+1}^{\mu+d} \gamma_j < \gamma^*$ **then**
  **return** 0
**for** $j = 1, \ldots, d$ **do**
  $s_j \leftarrow \mathsf{refresh}(\mathsf{seed}, s_{j-1}, \mathrm{I}_{\mu+j})$
$(s_0^*, r_0^*) \leftarrow \boxed{\mathsf{next}(\mathsf{seed}, s_d, \ell)}$
$\boxed{s_1^* \overset{\$}{\leftarrow} \mathsf{M}_{\mathsf{R}}(s_0^*)}$
**if** $r_0^* = \perp$ **then**
  $r_1^* \leftarrow \perp$
**else**
  $r_1^* \overset{\$}{\leftarrow} \{0,1\}^{\boxed{\ell}}$
$\mathsf{b}^* \overset{\$}{\leftarrow} \mathcal{A}(\sigma', s_{\mathsf{b}}^*, r_{\mathsf{b}}^*, \mathrm{I}_{\mu+d+1}, \ldots, \mathrm{I}_{\mathsf{q}_{\mathcal{D}}})$
**if** $\mathsf{b}^* = \mathsf{b}$ **then**
  **return** 1
**else**
  **return** 0

$\underline{\mathsf{Preserve}(\mathsf{G}, \mathcal{A}, \mathsf{M}, \ell)}$
$(\mathsf{seedgen}, \mathsf{setup}, \mathsf{refresh}, \mathsf{next}, \mathsf{tick}) \leftarrow \mathsf{G}$
$\mathsf{seed} \leftarrow \mathsf{seedgen}; \mathsf{b} \overset{\$}{\leftarrow} \{0,1\}$
$(s_0', \mathrm{I}_1, \ldots, \mathrm{I}_d, \sigma') \leftarrow \mathcal{A}(\mathsf{seed})$
$\boxed{s_0 \overset{\$}{\leftarrow} \mathsf{M}_{\mathsf{P}}(s_0')}$
**for** $j = 1, \ldots, d$ **do**
  $s_j \leftarrow \mathsf{refresh}(\mathsf{seed}, s_{j-1}, \mathrm{I}_j)$
$(s_0^*, r_0^*) \leftarrow \boxed{\mathsf{next}(\mathsf{seed}, s_d, \ell)}$
$\boxed{s_1^* \overset{\$}{\leftarrow} \mathsf{M}_{\mathsf{P}}(s_0^*)}$
**if** $r_0^* = \perp$ **then**
  $r_1^* \leftarrow \perp$
**else**
  $r_1^* \overset{\$}{\leftarrow} \{0,1\}^{\boxed{\ell}}$
$\mathsf{b}^* \overset{\$}{\leftarrow} \mathcal{A}(\sigma', s_{\mathsf{b}}^*, r_{\mathsf{b}}^*)$
**if** $\mathsf{b}^* = \mathsf{b}$ **then**
  **return** 1
**else**
  **return** 0

$\underline{\mathsf{get\text{-}refresh}\ ()}$
$\mu \leftarrow \mu + 1$
**return** $\mathrm{I}_\mu$
```

Figure 4.7: Preserving and recovering security games for G outputting ℓ-bits with split masking function M. Boxes indicate changes from [29, Definitions 3 & 4].

4.5.1 Variable-Output Preserving Security

One of the main differences in the preserving security game is that unlike the original preserving security of [29], the initial state is controlled by the adversary, similar to recovering security. However, the split masking function is then applied before the game proceeds, which should yield an "ideal" state. We present the following definitions that describe variable-output preserving security and a witnessed version of the same, which utilises a split masking function to make the proof easier.

Definition 4.5.1 (Variable Output Preserving Security)**.** A VOPWI is said to have $(t, \ell_{\mathsf{max}}, \epsilon_{\mathsf{p}})$-variable output preserving security, if, for any adversary $\mathcal{A}$ running in time t, and for all $\ell_i \in [1, \ell_{\mathsf{max}}]$ the preserving advantage defined by

$$\mathsf{Adv}^{\mathsf{pres}}_{\mathsf{G}^{\ell_{\mathsf{max}}}}(\mathcal{A}) = 2\Pr\left[\mathsf{Preserve}(\mathsf{G}, \mathcal{A}, \ell_i) = 1\right] - 1,$$

satisfies $\mathsf{Adv}^{\mathsf{pres}}_{\mathsf{G}^{\ell_{\mathsf{max}}}}(\mathcal{A}) \leq \epsilon_{\mathsf{p}}$, for $\mathsf{Preserve}$ as in Figure 4.7.

Definition 4.5.2 ((Witnessed) Variable Output Preserving Security)**.** A VOPWI is said to have $(t, \ell_{\mathsf{max}}, \epsilon_{\mathsf{p}})$-variable output preserving security, witnessed by the split masking function M, if, for any adversary $\mathcal{A}$ running in time t, and for all $\ell_i \in [1, \ell_{\mathsf{max}}]$ the preserving advantage defined by

$$\mathsf{Adv}^{\mathsf{pres}}_{\mathsf{M};\mathsf{G}^{\ell_{\mathsf{max}}}}(\mathcal{A}) = 2\Pr\left[\mathsf{Preserve}(\mathsf{G}, \mathcal{A}, \mathsf{M}, \ell_i) = 1\right] - 1,$$

satisfies $\mathsf{Adv}^{\mathsf{pres}}_{\mathsf{M};\mathsf{G}^{\ell_{\mathsf{max}}}}(\mathcal{A}) \leq \epsilon_{\mathsf{p}}$ for $\mathsf{Preserve}$ as in Figure 4.7 with the masking function M_{P} set to return a string sampled uniformly at random.

4.5.2 Variable-Output Recovering Security

We present the following definitions that describe variable output recovering security and a witnessed version of the same.

Definition 4.5.3 (Variable Output Recovering Security)**.** A VOPWI is said to have $(t, \mathsf{q}_{\mathcal{D}}, \ell_{\mathsf{max}}, \gamma^*, \epsilon_{\mathsf{r}})$-variable output recovering security, if, for any adversary $\mathcal{A}$ and legitimate sampler $\mathcal{D}$, both running in time t, and, for all $\ell_i \in [1, \ell_{\mathsf{max}}]$, the recovering advantage defined by

$$\mathsf{Adv}^{\mathsf{rec}}_{\mathsf{G}^{\ell_{\mathsf{max}}}}(\mathcal{A}) = 2\Pr\left[\mathsf{Recover}(\mathsf{G}, \mathcal{A}, \mathcal{D}, \ell_i) = 1\right] - 1,$$

satisfies $\mathsf{Adv}^{\mathsf{rec}}_{\mathsf{G}^{\ell_{\mathsf{max}}}}(\mathcal{A}) \leq \epsilon_{\mathsf{r}}$ for $\mathsf{Recover}$ as in Figure 4.7.

Definition 4.5.4 ((Witnessed) Variable Output Recovering Security)**.** A VOPWI is said to have $(t, \mathsf{q}_{\mathcal{D}}, \ell_{\mathsf{max}}, \gamma^*, \epsilon_{\mathsf{r}})$-variable output recovering security, witnessed by the split masking function M, if, for any adversary $\mathcal{A}$ and legitimate sampler $\mathcal{D}$, both running in time t, and, for all $\ell_i \in [1, \ell_{\mathsf{max}}]$, the recovering advantage defined by

$$\mathsf{Adv}^{\mathsf{rec}}_{\mathsf{M};\mathsf{G}^{\ell_{\mathsf{max}}}}(\mathcal{A}) = 2\Pr\left[\mathsf{Recover}(\mathsf{G}, \mathcal{A}, \mathcal{D}, \mathsf{M}, \ell_i) = 1\right] - 1,$$

satisfies $\mathsf{Adv}^{\mathsf{rec}}_{\mathsf{M};\mathsf{G}^{\ell_{\mathsf{max}}}}(\mathcal{A}) \le \epsilon_{\mathsf{r}}$ for Recover as in Figure 4.7 with the masking function M_{R} set to return a string sampled uniformly at random.

4.5.3 Updated Robustness Notion

The changes introduced in Figure 4.5 culminate in the full variable-output robustness game. Instead of limiting the adversary by the number of get-next and next-ror queries, we will limit her based upon the total output of the generator.

This allows an adversary more freedom, for example, if a generator is preserving secure for ℓ_{max} then it may seem obvious that reducing the output size would only decrease the adversarial advantage. However, there could be an output value that may cause bad behaviour in the algorithm, such as outputting the first half of the updated state when ℓ is precisely half the size of the state. Varying the output size also allows an adversary to make more queries at the cost of output size, or receive a larger amount but at the cost of future calls. This more accurately represents the requests made to a PRNG in practice, where the amount of randomness requested by consuming applications varies.

Recall, $\mathsf{q}_{\mathcal{D}}$ is the number of calls an adversary is allowed to make to $\mathcal{D}-$refresh, and q_{S} is the number of calls an adversary is allowed to make to get-state and set-state.

Theorem 4.5.5 (Boostrapped Variable-Output Robustness).
Let ℓ_{total} be a positive integer and the total amount of output in bits an adversary $\mathcal{A}$ is allowed to request. Let q *be the total number of* next *calls in a particular iteration of* M-rob*, assuming the amount of output is less than or equal to ℓ_{total} and no single ℓ_i exceeds ℓ_{max}. If there is an idempotent split masking function* M *that witnesses the $(t, \ell_{max}, \epsilon_{\mathsf{p}})$-variable output preserving and $(t, \mathsf{q}_{\mathcal{D}}, \ell_{max}, \gamma^*, \epsilon_{\mathsf{r}})$-variable output recovering security of a* VOPWI G *and* M *is $(\mathsf{G}, \mathcal{D}, t, \epsilon_{\mathsf{h}})$-honest, then* G *is*

$$\left(t', \ell_{max}, , \ell_{total}, \mathsf{q}_{\mathcal{D}}, \mathsf{q}_{\mathsf{S}}, \gamma^*, \epsilon_{\mathsf{h}} + \max_{\substack{(\ell_1,\dots,\ell_{\mathsf{q}}):\\ \sum_{i=1}^{\mathsf{q}} \ell_i \le \ell_{total}}} \left(\sum_{i=1}^{\mathsf{q}} (\epsilon_{\mathsf{p}} + \epsilon_{\mathsf{r}}) \right) \right) \text{-variable-output robust.}$$

An example of the total amount of output an adversary is allowed to make would be to use the old system of $\ell_{\mathsf{total}} = \ell * \mathsf{q_S}$ for some normal maximum number of queries $\mathsf{q_S}$. When referring to a particular instance of the robustness game where an adversary requests output sizes $\ell_1, \ldots, \ell_{\mathsf{q}}$, we speak of $\mathcal{A}$'s requests in terms of a vector $(\ell_1, \ldots, \ell_{\mathsf{q}})$.

Proof. We proceed by partitioning the uncorrupted next queries into "preserving" and "recovering" queries. We use the term "next query" to refer to both next and next-ror oracle queries. If at any point the corrupt flag is set to "true", the next next query once the flag has been reset is called a "recovering query". When a recovering query is made, we associate with it a most recent entropy drain (MRED) query, which will be the previous get-state, set-state or get-next query. We assume without loss of generality that the adversary does not make a next query when the corrupt flag is set to true, since this will reset the entropy counter and always output real output. The leftover uncorrupted next queries are called "preserving queries".

Let M-rob be the robustness experiment where the initial state s_0 is overwritten with $\mathsf{M_I}(s_0)$. Next, define the game $\mathbf{G_i}$ to be the same as M-rob with the following changes, assuming the ith next query is uncompromised:

- In the first i next queries, next-ror replaces the updated state with a masked version $\mathsf{M_P}(s_{i+1})$ (or $\mathsf{M_R}(s_{i+1})$ if it is a recovering call) and always returns r_1.
- Similarly, in the first i next queries, get-next replaces the updated state with masked version $\mathsf{M_P}(s_{i+1})$ (or $\mathsf{M_R}(s_{i+1})$ if it is a recovering call) and overwrites the output of the get-next query with $r \xleftarrow{\$} \{0,1\}^{\ell_j}$.

Further, define $\mathbf{G_{i+1/2}}$ which behaves precisely the same as $\mathbf{G_{i+1}}$ when the $(i+1)$-st next query is a preserving next query, by replacing the $i+1$st output with a random string, and the updated state $\mathsf{M_P}(s_{i+2})$. For all other next queries, i.e. the recovering queries, $\mathbf{G_{i+1/2}}$ behaves like $\mathbf{G_i}$.

The edge cases here are as follows

$$\Pr\left[\mathbf{G_0}(\mathsf{G}, \mathcal{A}, \mathcal{D}, \ell_{\mathsf{max}}, \ell_{\mathsf{total}}) = 1\right] \leq \Pr\left[\mathsf{M\text{-}rob}(\mathsf{G}, \mathcal{A}, \mathcal{D}, \ell_{\mathsf{max}}, \ell_{\mathsf{total}}) = 1\right] + \epsilon_{\mathsf{h}}, \quad (4.5.1)$$

from the properties of the masking function, and

$$\Pr\left[\mathbf{G_q}(\mathsf{G},\mathcal{A},\mathcal{D},\ell_{\mathsf{max}},\ell_{\mathsf{total}})=1\right]=\frac{1}{2}, \tag{4.5.2}$$

since all outputs are independent of the choice of the bit b. This leaves us with

$$\begin{aligned}&\Pr[\mathsf{M\text{-}rob}(\mathsf{G},\mathcal{A},\mathcal{D},\ell_{\mathsf{max}},\ell_{\mathsf{total}})=1]\\ &\quad\leq\left(\sum_{i=1}^{q-1}\left|\Pr\left[\mathbf{G_i}(\mathsf{G},\mathcal{A},\mathcal{D},\ell_{\mathsf{max}},\ell_{\mathsf{total}})=1\right]-\Pr\left[\mathbf{G_{i+1/2}}(\mathsf{G},\mathcal{A},\mathcal{D},\ell_{\mathsf{max}},\ell_{\mathsf{total}})=1\right]\right|\right.\\ &\qquad\left.+\left|\Pr\left[\mathbf{G_{i+1/2}}(\mathsf{G},\mathcal{A},\mathcal{D},\ell_{\mathsf{max}},\ell_{\mathsf{total}})=1\right]-\Pr\left[\mathbf{G_{i+1}}(\mathsf{G},\mathcal{A},\mathcal{D},\ell_{\mathsf{max}},\ell_{\mathsf{total}})=1\right]\right|\right).\end{aligned}$$

The proof follows very closely to the original proof [29] and the adapted proof of [55] by proceeding with the following two lemmas:

Lemma 4.5.6.
Let $\mathbf{G_i},\mathbf{G_{i+1/2}}$ *be defined as above and let* G *be a* $(t,\ell_{max},\epsilon_{\mathsf{p}})$*-variable output preserving secure* VOPWI*, witnessed by the split masking function* M*. Then for any adversary* $\mathcal{A}$*,*

$$\left|\Pr\left[\mathbf{G_i}(\mathsf{G},\mathcal{A},\mathcal{D},\ell_{max},\ell_{total})=1\right]-\Pr\left[\mathbf{G_{i+1/2}}(\mathsf{G},\mathcal{A},\mathcal{D},\ell_{max},\ell_{total})=1\right]\right|\leq\epsilon_{\mathsf{p}}.$$

Proof. Assume an adversary $\mathcal{A}$ exists, we will construct from $\mathcal{A}$ a new adversary $\mathcal{B}$ for the preserving security game. Also assume that the $(i+1)$st next query is a preserving query, since otherwise $\mathbf{G_i}$ and $\mathbf{G_{i+1/2}}$ are identical. We present the construction and explanation in Figure 4.8.

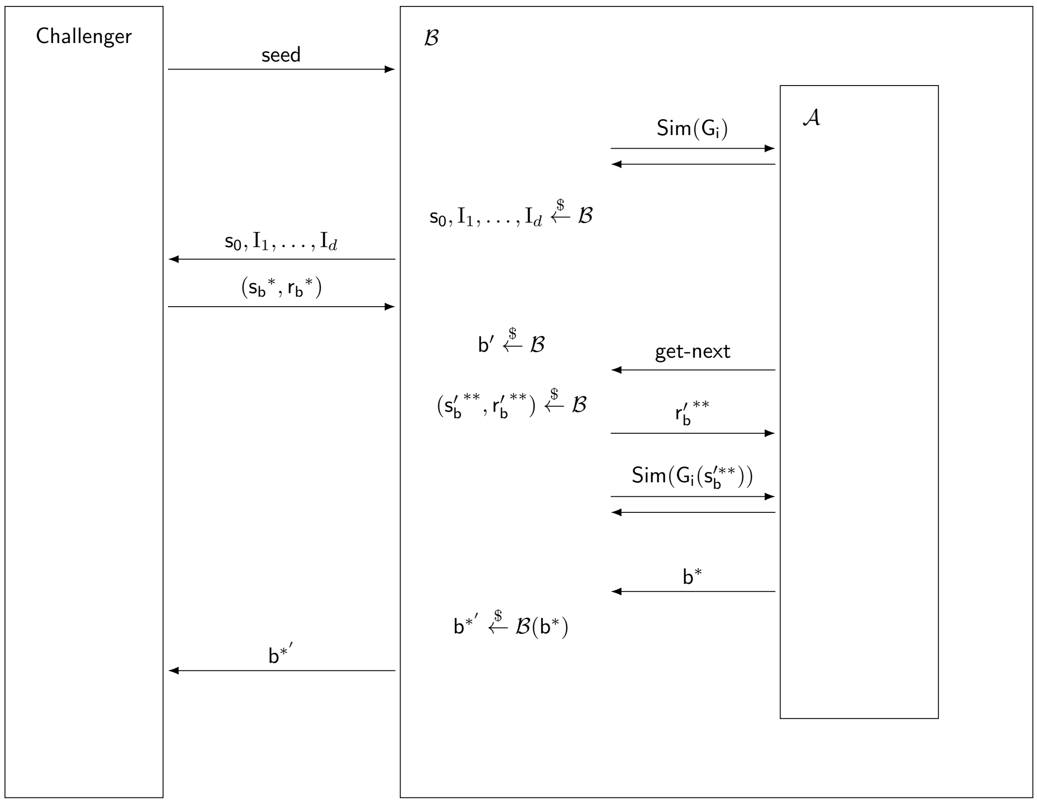

Challenger
seed
$\mathcal{B}$
$\mathcal{A}$
Sim(G$_i$)
$s_0, I_1, \ldots, I_d \xleftarrow{\$} \mathcal{B}$
$s_0, I_1, \ldots, I_d$
$(s_b{}^*, r_b{}^*)$
$b' \xleftarrow{\$} \mathcal{B}$
get-next
$(s_b'^{**}, r_b'^{**}) \xleftarrow{\$} \mathcal{B}$
$r_b'^{**}$
Sim(G$_i$($s_b'^{**}$))
b^*
$b^{*\prime} \xleftarrow{\$} \mathcal{B}(b^*)$
$b^{*\prime}$

Figure 4.8: The construction of preserving security adversary $\mathcal{B}$ from adversary $\mathcal{A}$.

1. To begin, $\mathcal{B}$ is given seed by the challenger, and uses it, along with knowledge of $\mathcal{D}$ to simulate $\mathbf{G_i}$ for $\mathcal{A}$ until the $i+1$st next query.

2. Let s_0 be the resulting state from this simulation that $\mathcal{B}$ uses for the $i+1$st next query.

3. $\mathcal{B}$ generates entropy inputs $I_1, \ldots, I_d$ and passes them, along with s_0 to the challenger.

4. The challenger replies with (s^*_b, r^*_b), and $\mathcal{B}$ flips its own bit, $b' \xleftarrow{\$} \{0,1\}$.

5. On receipt of $\mathcal{A}$'s $(i+1)$st next query, $\mathcal{B}$ calculates the following

$$(s^{**}_{b'}, r^{**}_{b'}) =: \begin{cases} (s^{**}_0, r^{**}_0) = (s^*_b, r^*_b) & \text{for } b' = 0, \\ (s^{**}_1, r^{**}_1) = \left(\mathsf{M_P}(s^*_b), \begin{cases} \perp & \text{if } r^*_b = \perp, \\ r^{**}_1 & \text{otherwise} \end{cases} \right) & \text{for } b' = 1. \end{cases}$$

for $r^{**}_1 \xleftarrow{\$} \{0,1\}^{\ell_{i+1}}$, and returns $r^{**}_{b'}$ to $\mathcal{A}$.

6. Next, $\mathcal{B}$ continues simulating the remainder of the game for $\mathcal{A}$ using $s^{**}_{b'}$.

7. When $\mathcal{A}$ outputs b^*, $\mathcal{B}$ outputs $\mathsf{b}^{*'} = 1$ if $\mathsf{b}^* = \mathsf{b}'$ or $\mathsf{b}^{*'} = 0$ otherwise.

We now walk through the different possibilities:

First, if the original challenge bit $\mathsf{b} = 0$, then $\mathcal{B}$ has exactly simulated $\mathbf{G_i}$ for $\mathcal{A}$; the first i next queries followed the definition of $\mathbf{G_i}$ and the $(i+1)$st next query returned the "real" values if $\mathsf{b}' = 0$, and returned a mask of the state and random bits if $\mathsf{b}' = 1$. Since M is idempotent, the distribution of s_0, taken from the output of the last next query, is not changed when the challenger applies a mask to it in line 4 of the preserving game in Figure 4.7. $\mathbf{G_i}$ requires the state be masked after every next query and so s_0 is essentially already masked.

Secondly, if the original challenge bit $\mathsf{b} = 1$, then regardless of b', $\mathcal{B}$ has exactly simulated $\mathbf{G_{i+1/2}}$ for $\mathcal{A}$; since it gives $\mathcal{A}$ a mask of the state and uniformly random bits for the $(i+1)$st query. In the case where $\mathsf{b}' = 0$, $\mathcal{A}$ passed $\mathsf{M_P}(s_0^*)$ to $\mathcal{B}$ along with random output, which are passed straight onto $\mathcal{A}$. When $\mathsf{b}' = 1$, $\mathcal{B}$ masks the state it was given by the challenger, resulting in $\mathsf{M_P}(\mathsf{M_P}(s_0^*))$, which by the idempotence of M is identically distributed to $\mathsf{M_P}(s_0^*)$, it picks a new output string uniformly at random and passes both to $\mathcal{A}$. Lastly, the above gives us that

$$\begin{aligned}
&\left|\Pr\left[\mathbf{G_i}(\mathcal{A}, \mathcal{D}, \ell_{\mathsf{max}}, \ell_{\mathsf{total}}) = 1\right] - \Pr\left[\mathbf{G_{i+1/2}}(\mathcal{A}, \mathcal{D}, \ell_{\mathsf{max}}, \ell_{\mathsf{total}}) = 1\right]\right| \\
&= \left|\Pr\left[\mathsf{b}' = \mathsf{b}^* \mid \mathsf{b} = 0\right] - \Pr\left[\mathsf{b}' = \mathsf{b}^* \mid \mathsf{b} = 1\right]\right| \\
&= \left|2\Pr\left[\mathsf{b}^{*'} = \mathsf{b}\right] - 1\right| \leq \epsilon_{\mathsf{p}}.
\end{aligned}$$

□

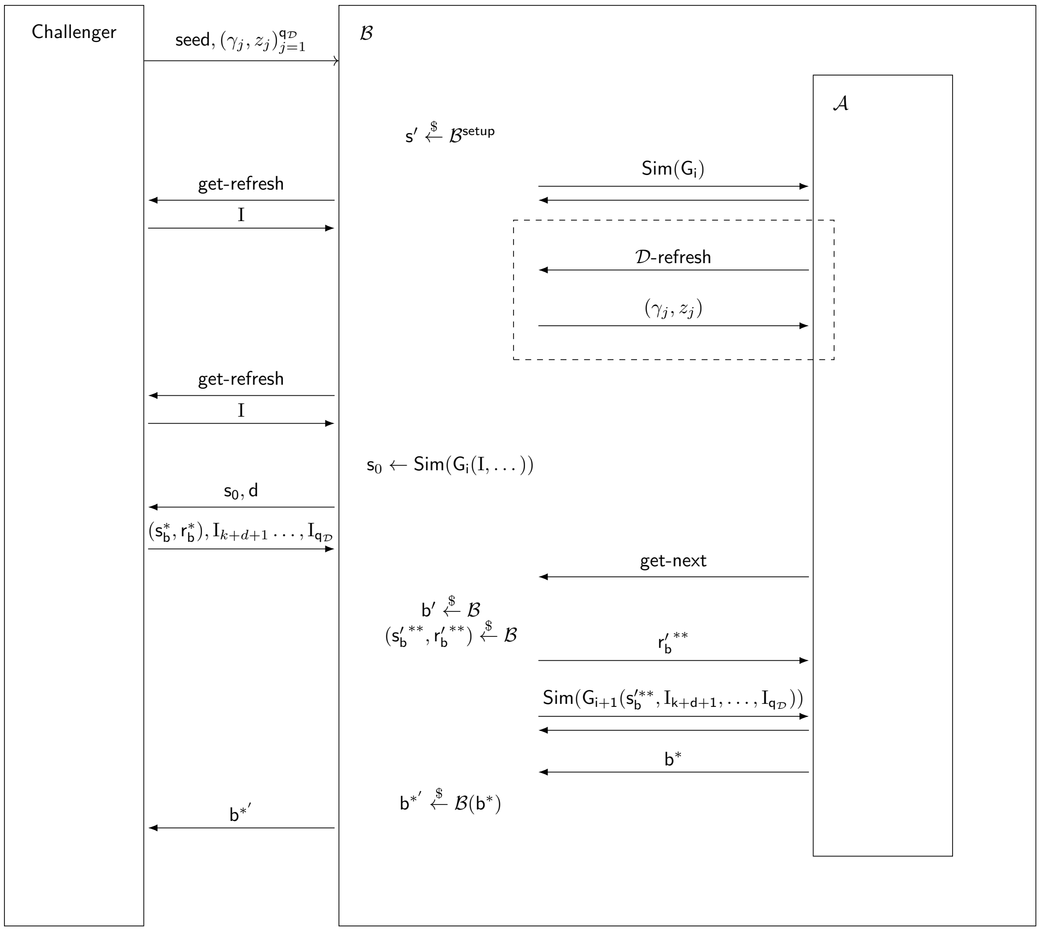

Challenger
seed, $(\gamma_j, z_j)_{j=1}^{q_{\mathcal{D}}}$
$\mathcal{B}$
$\mathcal{A}$
$s' \xleftarrow{\$} \mathcal{B}^{setup}$
$Sim(G_i)$
get-refresh
I
$\mathcal{D}$-refresh
(γ_j, z_j)
get-refresh
I
$s_0 \leftarrow Sim(G_i(I, \dots))$
s_0, d
$(s_b^*, r_b^*), I_{k+d+1} \dots, I_{q_{\mathcal{D}}}$
get-next
$b' \xleftarrow{\$} \mathcal{B}$
$(s_b'^{**}, r_b'^{**}) \xleftarrow{\$} \mathcal{B}$
$r_b'^{**}$
$Sim(G_{i+1}(s_b'^{**}, I_{k+d+1}, \dots, I_{q_{\mathcal{D}}}))$
b^*
$b^{*'} \xleftarrow{\$} \mathcal{B}(b^*)$
$b^{*'}$

Figure 4.9: The construction of the recovering security adversary $\mathcal{B}$ from adversary $\mathcal{A}$.

Lemma 4.5.7.

Let $\mathbf{G_{i+1/2}}, \mathbf{G_{i+1}}$ *be defined as above and let* G *be a* $(t, q_{\mathcal{D}}, \ell_{max}, \gamma^*, \epsilon_r)$*-variable output recovering secure* VOPWI*, witnessed by the split masking function* M*, then for any adversary* $\mathcal{A}$*,*

$$\left|\Pr\left[\mathbf{G_{i+1/2}}(\mathsf{G}, \mathcal{A}, \mathcal{D}, \ell_{max}, \ell_{total}) = 1\right] - \Pr\left[\mathbf{G_{i+1}}(\mathsf{G}, \mathcal{A}, \mathcal{D}, \ell_{max}, \ell_{total}) = 1\right]\right| \leq \epsilon_r^{\ell_{i+1}}.$$

Proof. Assume an adversary $\mathcal{A}$ exists, we will construct from $\mathcal{A}$ a new adversary $\mathcal{B}$ for the recovering security game. Similarly to Lemma 4.5.6, assume that the $(i+1)$st next query is a recovering query. We present the construction in Figure 4.9 and explain the steps below.

1. To begin, $\mathcal{B}$ is given seed and information relating to the refresh values $(\gamma_j, zj)_{j=1}^{\mathsf{q}_{\mathcal{D}}}$ by the challenger, and generates an initial state $s' \xleftarrow{\$} \mathsf{setup}$.

2. $\mathcal{B}$ simulates $\mathbf{G_i}$ for $\mathcal{A}$, providing $\mathcal{A}$ with seed and using the leaked information it received and its $\mathsf{get\text{-}refresh}$ oracle to simulate the $\mathcal{D}-\mathsf{refresh}$ oracle for $\mathcal{A}$.

3. After the ith next query $\mathcal{B}$, in response to a $\mathcal{D}-\mathsf{refresh}$ query from $\mathcal{A}$ will only return the associated pair (γ_j, z_j) it received from the challenger, it does not update the state.

4. On the $(i+1)$st next query, $\mathcal{B}$ calls its $\mathsf{get\text{-}refresh}$ oracle to update the state to the point immediately following the MRED to the recovering query, and sets this as s_0.

5. $\mathcal{B}$ counts the number of $\mathcal{D}-\mathsf{refresh}$ calls $\mathcal{A}$ made after the MRED and submits (s_0, d) to the challenger.

6. $\mathcal{B}$ receives the challenge $(s^*_{\mathsf{b}}, r^*_{\mathsf{b}})$ and the remaining entropy strings $\mathrm{I}_{k+d+1}, \ldots, \mathrm{I}_{\mathsf{q}_{\mathcal{D}}}$ which it uses later to continue the simulation.

7. On receipt of $\mathcal{A}$'s $(i+1)$st next query, $\mathcal{B}$ flips its own bit $\mathsf{b}' \xleftarrow{\$} \{0,1\}$ and calculates the following

$$(s^{**}_{\mathsf{b}'}, r^{**}_{\mathsf{b}'}) = \begin{cases} (s^{**}_0, r^{**}_0) = (s^*_{\mathsf{b}}, r^*_{\mathsf{b}}) & \text{for } \mathsf{b}' = 0, \\ (s^{**}_1, r^{**}_1) = \left(\mathsf{M_R}(s^*_{\mathsf{b}}), \begin{cases} \perp & \text{if } r^*_{\mathsf{b}} = \perp, \\ r^{**}_1 \xleftarrow{\$} \{0,1\}^{\ell_{i+1}} & \text{otherwise} \end{cases} \right) & \text{for } \mathsf{b}' = 1. \end{cases}$$

   and returns $r^{**}_{\mathsf{b}'}$ to $\mathcal{A}$.

8. Lastly, $\mathcal{B}$ uses the remaining entropy strings $\mathrm{I}_{k+d+1}, \ldots, \mathrm{I}_{\mathsf{q}_{\mathcal{D}}}$ to continue simulating $\mathbf{G_{i+1}}$ for $\mathcal{A}$ from the state $s^{**}_{\mathsf{b}'}$.

9. When $\mathcal{A}$ outputs b^*, $\mathcal{B}$ outputs $\mathsf{b}^{*'} = 1$ if $\mathsf{b}^* = \mathsf{b}'$ or $\mathsf{b}^{*'} = 0$ otherwise.

We now walk through the different possibilities:

First, if the original challenge bit $\mathsf{b} = 0$, then $\mathcal{B}$ has exactly simulated $\mathbf{G_{i+1/2}}$ for $\mathcal{A}$; the first i next queries followed the definition of $\mathbf{G_i}$ and the $(i+1)$st query returned the "real" values if $\mathsf{b}' = 0$, and returned a mask of the state and uniformly random bits if $\mathsf{b}' = 1$.

Secondly, if the original challenge bit $\mathsf{b} = 1$, then regardless of b', $\mathcal{B}$ has exactly simulated $\mathbf{G_{i+1}}$ for $\mathcal{A}$; since it gives $\mathcal{A}$ a mask of the state and uniformly random bits for the $(i+1)$st query, again by using the property that $\mathsf{M_R}(\mathsf{M_R}(s))$ is identically distributed as $\mathsf{M_R}(s)$.

Lastly, the above gives us that

$$\begin{aligned}
&\left|\Pr\left[\mathbf{G_{i+1/2}}(\mathcal{A}, \mathcal{D}, \ell_{\mathsf{max}}, \ell_{\mathsf{total}}) = 1\right] - \Pr\left[\mathbf{G_{i+1}}(\mathcal{A}, \mathcal{D}, \ell_{\mathsf{max}}, \ell_{\mathsf{total}}) = 1\right]\right| \\
&= \left|\Pr\left[\mathsf{b}' = \mathsf{b}^* \mid \mathsf{b} = 0\right] - \Pr\left[\mathsf{b}' = \mathsf{b}^* \mid \mathsf{b} = 1\right]\right| \\
&= \left|2\Pr\left[\mathsf{b}^{*'} = \mathsf{b}\right] - 1\right| =\leq \epsilon_{\mathsf{r}}.
\end{aligned}$$

□

Putting these two lemmas together, with eqs. (4.5.1) and (4.5.2), while taking the maximum over the possible combinations of $(\ell_1, \ldots, \ell_{\mathsf{q}})$ (which determines each q), yields the result. □

4.6 Constructing a VOPWI from a PWI

The natural progression of our extension is a way to construct a secure VOPWI from a secure PWI. In this section we briefly investigate several propositions for building a VOPWI from a PWI, stating if they are achievable or justifying why they are not.

Let us assume that the refresh algorithm of the PWI fulfils the new requirements of the setup algorithm. Then what remains is to consider the different modifications we can make to the next function. This in itself can be broken down into four possibilities:

1. Redesign the next algorithm of the PWI by adding a generate algorithm; this will be specific to the PWI design, and thus cannot be generalised.
2. Modify the next algorithm of the PWI to return output of a length $\ell_i \leq \ell$, maintaining the restriction on the number of calls to the next algorithm. Since this is essentially reducing the amount of output a PWI adversary receives, this is trivial.

3. Add a generate algorithm that takes as input the raw PWI output and expands it as required. This can easily be achieved with a suitably secure PRG.
4. Similar to the second item, modify the next algorithm of the PWI to return output of a length $\ell_i \leq \ell$, but maintain the total amount of output an adversary may receive as $\ell_{\mathsf{total}} = \mathsf{q}\ell$. This is the most complex proposition.

In the fourth item we described changing the restrictions on output of a PWI to the total amount of output restriction of a VOPWI. We assert that this cannot be done in general without restrictions based upon properties of the PWI. This is motivated by the following example:

Given a secure PWI, which is secure, up to a maximum of q calls to the next algorithm, else the PWI repeats its outputs in order. We construct a VOPWI from this PWI, and an adversary that will distinguish its output from random. The adversary requests maximal output on the first call to next, but the minimum amount (say 1 bit) from the following $\mathsf{q} - 1$ outputs. The adversary then requests the maximum amount of output on the $\mathsf{q} + 1$st call to the next algorithm, which, will be equal to the first output by the properties of the underlying PWI, and thus is distinguishable from random through being equal to the first output the adversary saw. This cannot be fixed by reducing the amount of output the adversary may request, but could be captured and mitigated by adding a reseed counter, similar to practical generators.

4.7 Conclusion

In this chapter we extended the security models of [29, 55] to allow a generator to output varying amounts of output, possibly utilising a sub-procedure that does a “small” state update between blocks of output. We also added a seedgen algorithm and expanded the setup algorithm to capture the initial state generation more accurately.

Chapter 5

Analysis of NIST Generators

Contents

In this chapter we make use of the security model developed in Chapter 4 to analyse the robustness of the NIST generators.

5.1 Preliminaries

The NIST special publications on random bit generation are SP800-90A [7], SP800-90B [9], and SP800-90C [8]. Document SP800-90A details specifications of several deterministic random bit generators (which we call PRNGs with input or PWIs). The SP800-90B document details recommendations for entropy sources used in conjunction with the PWIs detailed in SP800-90A. Lastly, SP800-90C is a comprehensive document on implementing the PWIs from document A and the entropy sources from document B securely.

We will be investigating the security of the hash_drbg and ctr_drbg in the security model defined in Chapter 4. We do not investigate the security of the hmac_drbg since there are papers such as [38, 58] that investigate the security claims for this generator.

In the NIST specification [7], there are several parameter and optional-extra choices for each DRBG design. This includes, but is not limited to the use of a derivation function to "mix" inputs into states. We will not be analysing each combination of choices, but will make our choices clear, along with justifying them.

We proceed in this chapter by defining some notation shared between the generators, followed by the specification of the hash_drbg and ctr_drbg in the format of our VOPWI definition (Definition 4.2.1). We follow this with a formal analysis of both generators in our security model in Chapter 4.

5.1.1 NIST Seed Structure

Here we summarise the different parts of what we will treat as the seed of the generator.

The nonce nonce, discussed in [7, Section 8.6.7] used in the NIST generators is not required to be kept secret, but is required to have either "at least $\lambda/2$ bits of entropy, or does not repeat for at least $\lambda/2$ bits". One example can be built from a time-stamp and monotonically increasing sequence. The nonce is used in the setup algorithm.

The personalisation string perstring is an optional but recommended input to the setup algorithm; it is primarily used to separate the instantiation from others, though knowledge of the personalisation string by the adversary should not degrade the security strength of the DRBG.

The additional input add is an optional input provided to the refresh and next algorithms. Knowledge of the additional input should not degrade the security strength of the generator. It is suggested in the specification that applications requesting randomness may provide add for each request, which could be used to separate generator next calls or provide a small amount of entropy.

The security field λ is an optional input in the setup algorithm for both the hash_drbg and ctr_drbg. Since the nonce nonce and additional input add are optional, the entropy of the initial state of the generator may rely entirely on the entropy input used by setup.

5.1.2 General NIST Notation

We start by describing in Table 5.1 some general notation applicable to both generators, including updates that we have made to port the NIST notation to our own. We do this for ease of reading and for later applying the security model to the generators.

We include generator specific notation at the beginning of each section to avoid a notation overload.

Our Notation	NIST Notation	Size (bits)	Description
hash_drbg	hash_drbg		The NIST generator based on an approved hash function.
ctr_drbg	ctr_drbg		The NIST generator based on an approved block cipher.
λ	security_strength		Advertised security strength of the generator.
n	seed_length		Size of different sub-states of the generators in bits.
ℓ_i	requested_number _of_bits		Requested number of bits from the generate process of the hash_drbg and ctr_drbg.
0xab	0xab		Hexadecimal notation of a byte.
s_i	Working_State		State of the generators.
s_0	seed		Initial state of a generator, either after initialisation (and reseeding in the NIST notation).
I_i	entropy_input	p	The entropy input used to refresh the state of a generator.
r_i	returned_bits	ℓ	The output of the PWI produced from state $i-1$.
nonce	nonce		A generated nonce for input into the setup algorithm.
perstring	personalization_s	$\leq 2^{35}$	An optional string used in the setup algorithm.
add	additional_input	$\leq 2^{35}$	An optional string used in the refresh algorithm.
seed	(nonce, perstring, add)		The optional strings.
$\mathsf{Len}(A)$	$\mathsf{Len}(A)$	$\lvert A\rvert$	The length in bits of the string A.
$\mathsf{L}_a(A)$	leftmost(A,a)		Left most a bits of A.
$\mathsf{R}_a(A)$	rightmost(A,a)		Right most a bits of A.
$\mathsf{Sel}_a^b(A)$	select(A,a,b)		Select from A, bits a to b.

Table 5.1: The notation used in [7], updated for continuity with our own notation.

5.2 The hash_drbg

5.2.1 Notation

Table 5.2 describes the additional notation required for the hash_drbg. The state of the hash_drbg $s : (v, c, \mathsf{rc})$ is made up of the v-state v which is frequently updated and used to produce output, and the c-state c which is supposed to retain the entropy from the previous entropy input and to separate outputs between refreshes.

The NIST specification states that the maximum number of calls between refreshes shall be at maximum 2^{48}. Since the reseed counter rc is initialised to 1, it will reach $1 + 2^{48}$ which requires 49 bits to represent.

Our Notation	NIST Notation	Size (bits)	Description
hash_drbg	HASH_DRBG		The NIST generator based on an approved hash function.
hash_df	Hash_df		An auxiliary algorithm used in the hash_drbg.
hashgen	Hashgen		An auxiliary algorithm used in the hash_drbg.
n_{H}	outlen	variable	Output length of the hash function.
ℓ_{max}		2^{19}	Maximum requested number of bits from the generate algorithm.
v_i	V	n	Sub-state of the working state.
c_i	C	n	Sub-state of the working state.
rc	reseed_counter	49	Number of next calls since last refresh, maximum 2^{48}.
$s_i := (v_i, c_i, \mathsf{rc})$	Working_State	$2n + 49$	i-th working state of the generator in full.

Table 5.2: Notation for the hash_drbg, updated for continuity with our notation.

	SHA-1	SHA-224 & SHA-512/224	SHA-256 & SHA-512/256	SHA-384	SHA-512
n_{H}	160	224	256	384	512
n	440	440	440	888	888
perstring	$\leq 2^{35}$				
add	$\leq 2^{35}$				
p	$\leq 2^{35}$				
ℓ_{max}	$\leq 2^{19}$ bits				
Requests between reseeds	$\leq 2^{48}$				

Table 5.3: Parameters of the different hash_drbg instantiations.

5.2.2 Specification of the Generator

Definition 5.2.1. In addition to the notation given in Table 5.2 and parameters in Table 5.3, let H be a hash function from $\{0,1\}^* \longrightarrow \{0,1\}^{n_{\mathsf{H}}}$. Let

$$n \approx 2n_{\mathsf{H}},$$
$$\lambda \leq p \leq 2^{35},$$
$$\ell_i \leq \ell_{\mathsf{max}} := 2^{19},$$
$$\mathsf{IFace} := \{0\},$$
$$(\mathsf{nonce}, \mathsf{perstring}, \mathsf{add}) \in \mathsf{Seedspace} := \{0,1\}^* \times \{0,1\}^{\leq 2^{35}} \times \{0,1\}^{\leq 2^{35}},$$
$$(v_i, c_i, \mathsf{rc}) \in \mathsf{Statespace} := \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^{49}.$$

The value of IFace is chosen since we are only using a single interface and will omit it from the rest of the specification. The hash_drbg algorithms are specified in Figure 5.2, making use of two auxiliary algorithms given in Figure 5.1.

```
hash_df($x, \ell$)
$y \leftarrow null$
$m \leftarrow \lceil \ell / n_{\mathsf{H}} \rceil$
$ctr \leftarrow 0x01$
for $i = 1$ to $m$
    $y \leftarrow y \| \mathsf{H}(ctr \| \ell \| x)$
    $ctr \leftarrow ctr + 1$
endfor
return $\mathsf{L}_{\ell}(y)$
```

```
hashgen($v_i, \ell_{i+1}$)
$m \leftarrow \lceil \ell_{i+1} / n_{\mathsf{H}} \rceil$
$W \leftarrow null$
for $j = 1$ to $m$
    $w \leftarrow \mathsf{H}(v_i)$
    $W \leftarrow W \| w$
    $v'_i \leftarrow (v_i + 1) \mod 2^n$
endfor
return $(v'_i, \mathsf{L}_{\ell_{i+1}}(W))$
```

Figure 5.1: The auxiliary algorithms hash_df and hashgen of hash_drbg.

```
hash_drbg.setup(seed, $(\mathrm{I}_0, \gamma_0, z_0)$)
parse seed as (nonce, perstring, add)
$v_0 \leftarrow$ hash_df$((\mathrm{I}_0 \| \mathsf{nonce} \| \mathsf{perstring}), n)$
$c_0 \leftarrow$ hash_df$((0x00 \| v_0), n)$
rc $\leftarrow 1$
return $s_0 = (v_0, c_0, \mathsf{rc})$
```

```
hash_drbg.seedgen
nonce $\xleftarrow{\$} \{0,1\}^*$
perstring $\xleftarrow{\$} \{0,1\}^{\leq 2^{35}}$
add $\xleftarrow{\$} \{0,1\}^{\leq 2^{35}}$
seed $\leftarrow$ (nonce, perstring, add)
return seed
```

```
hash_drbg.next(seed, $s_i, \ell_{i+1}$)
parse seed as (nonce, perstring, add)
parse $s_i$ as $(v_i, c_i, \mathsf{rc})$
$w \leftarrow \mathsf{H}(0x02 \| v_i \| \mathsf{add})$
$v'_i \leftarrow (v_i + w) \mod 2^n$
$(v'_{i+1}, r_{i+1}) \leftarrow$ hashgen$(v'_i, \ell_{i+1})$
$u \leftarrow \mathsf{H}(0x03 \| v'_{i+1})$
$v_{i+1} \leftarrow (v'_{i+1} + u + c_i + \mathsf{rc}) \mod 2^n$
$c_{i+1} \leftarrow c_i$
rc $\leftarrow$ rc $+ 1$
return $(s_{i+1}, r_{i+1}) = ((v_{i+1}, c_{i+1}, \mathsf{rc}), r_{i+1})$
```

```
hash_drbg.refresh(seed, $s_i, \mathrm{I}$)
parse seed as (nonce, perstring, add)
parse $s_i$ as $(v_i, c_i, \mathsf{rc})$
s $\leftarrow 0x01 \| v_i \| \mathrm{I} \| \mathsf{add}$
$v_{i+1} \leftarrow$ hash_df$(\mathsf{s}, n)$
$c_{i+1} \leftarrow$ hash_df$((0x00 \| v_{i+1}), n)$
rc $\leftarrow 1$
return $(v_{i+1}, c_{i+1}, \mathsf{rc})$
```

```
hash_drbg.tick(seed, $s_i$)
$s_{i+1} \leftarrow s_i$
return $s_{i+1}$
```

Figure 5.2: The algorithms describing the behaviour of the hash_drbg following the format of Definition 4.2.1.

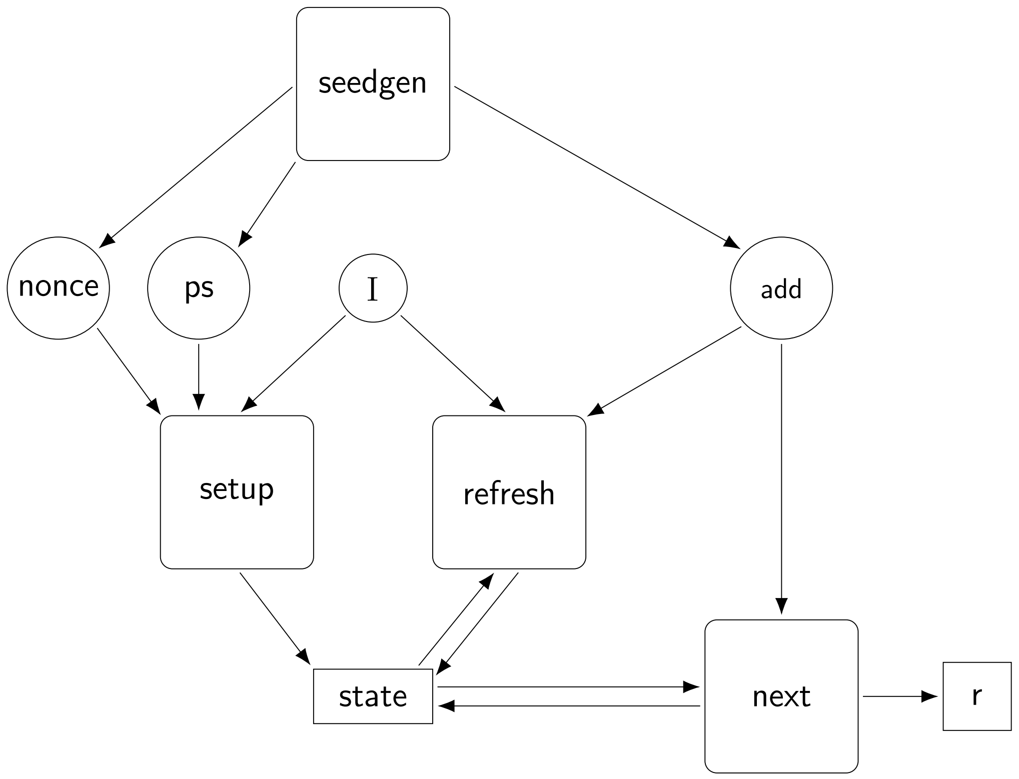
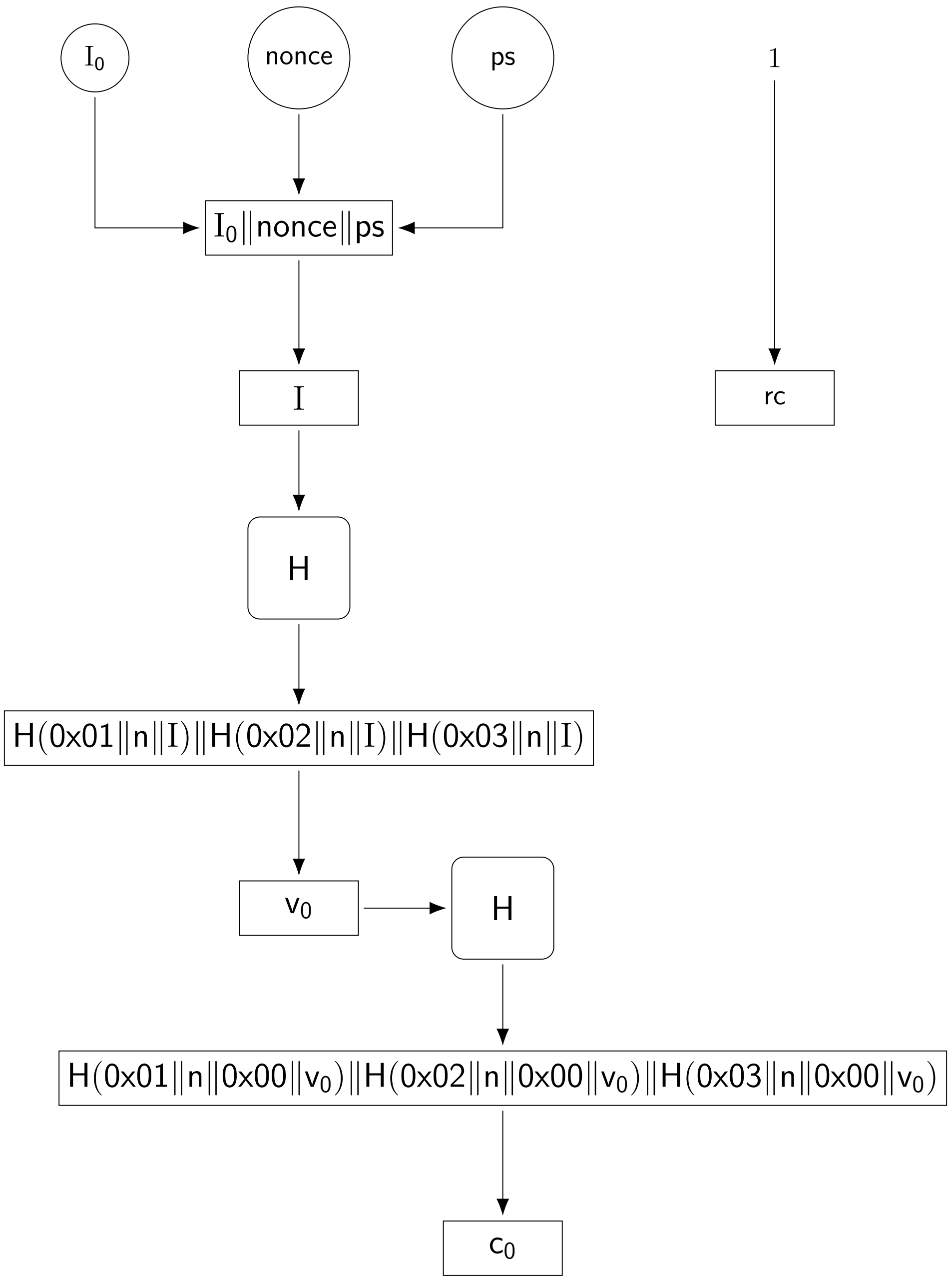


Figure 5.3: The hash_drbg algorithms and seed usage.

We now provide several overview diagrams of the different hash_drbg algorithms, including their subroutines for clarity. We use rounded edged boxes to indicate algorithms, such as the underlying block cipher, hash function or a subroutine. We use circles to represent entropy inputs including parts of the seed. We use rectangles to indicate states; a long rectangle split in two represents how a larger output is split into separate states. Lastly, we use squares to represent generator output.



I_0
nonce
ps
1
$I_0 \| \mathsf{nonce} \| \mathsf{ps}$
I
rc
H
$\mathsf{H}(0x01\|n\|I)\|\mathsf{H}(0x02\|n\|I)\|\mathsf{H}(0x03\|n\|I)$
v_0
H
$\mathsf{H}(0x01\|n\|0x00\|v_0)\|\mathsf{H}(0x02\|n\|0x00\|v_0)\|\mathsf{H}(0x03\|n\|0x00\|v_0)$
c_0

Figure 5.4: The hash_drbg setup algorithm.

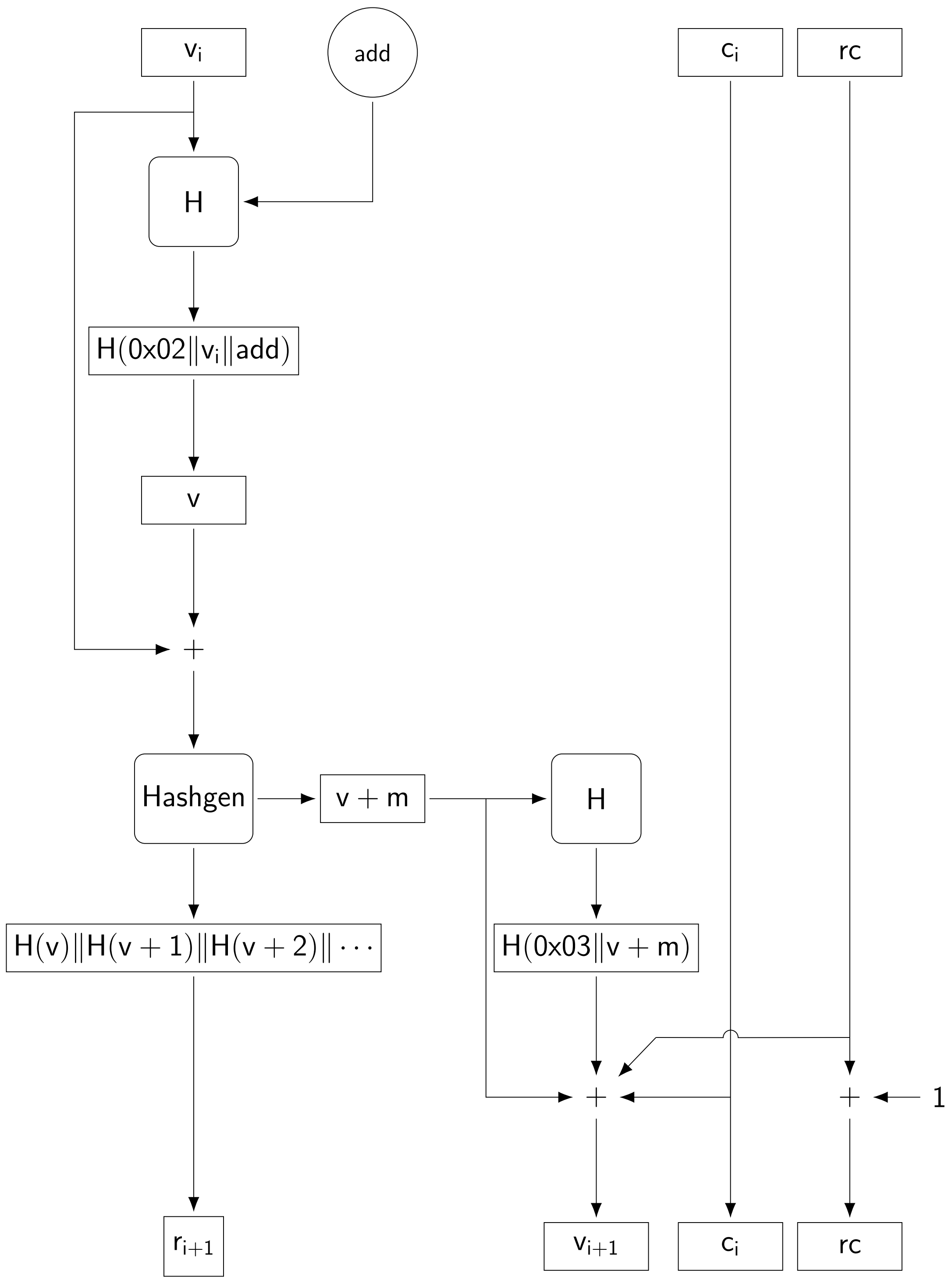


Figure 5.5: The hash_drbg next algorithm.

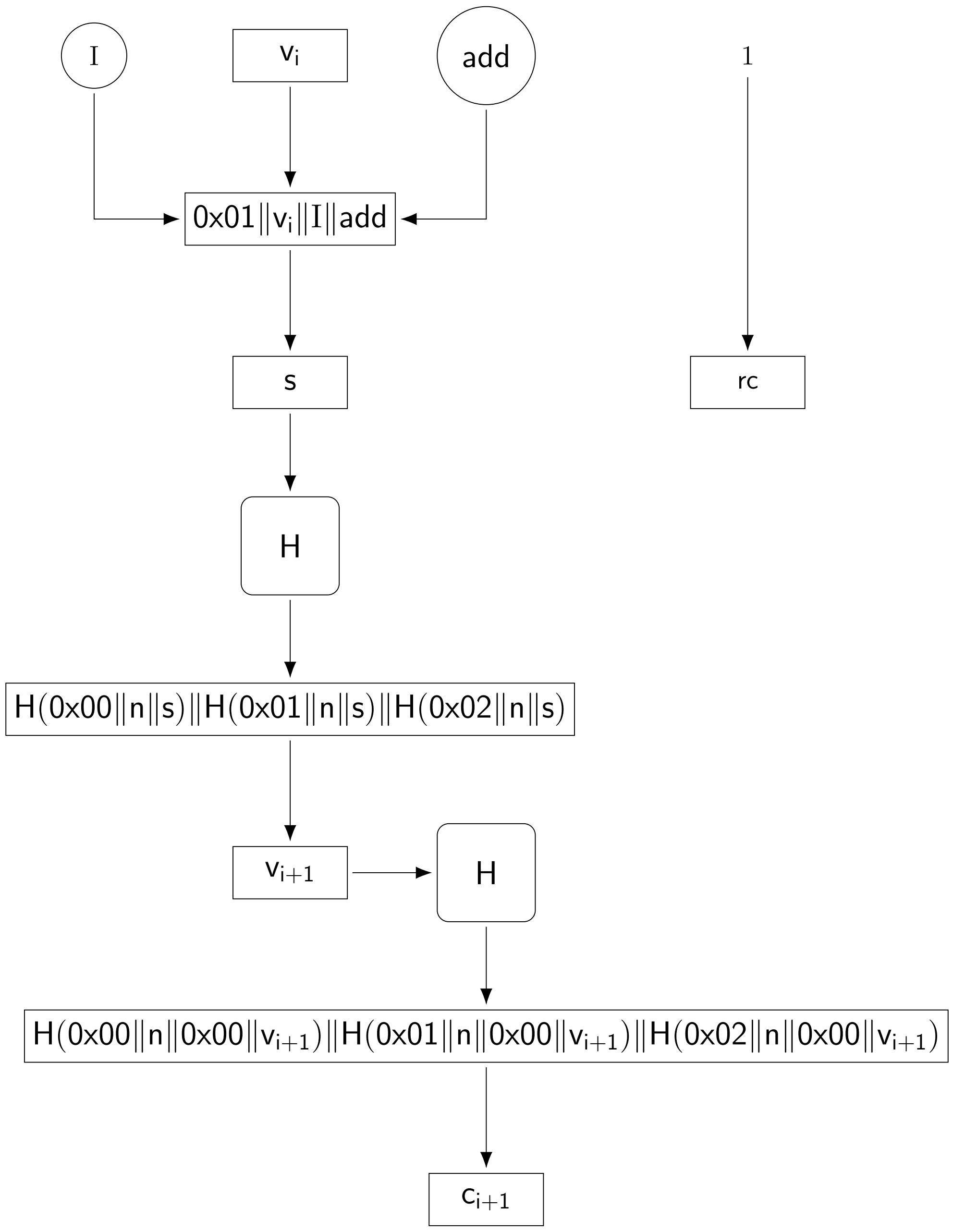

I
v_i
add
1
$0x01\|v_i\|I\|add$
s
rc
H
$H(0x00\|n\|s)\|H(0x01\|n\|s)\|H(0x02\|n\|s)$
v_{i+1}
H
$H(0x00\|n\|0x00\|v_{i+1})\|H(0x01\|n\|0x00\|v_{i+1})\|H(0x02\|n\|0x00\|v_{i+1})$
c_{i+1}

Figure 5.6: The hash_drbg refresh algorithm.

5.2.3 Algorithm Descriptions

hash_df The hash derivation function is an auxiliary algorithm of the hash_drbg. On input of a string x and required output length ℓ, the hash_df repeatedly hashes a counter concatenated with the required length and input string. The hash_df algorithm then returns a concatenation of the hash outputs, truncating as necessary. The algorithm is designed to either derive an internal state from the input given, or to distribute entropy throughout a bit string.

hashgen The hash generation algorithm is the second auxiliary algorithm of the hash_drbg. The hashgen algorithm takes as input a string v_i and requested number of output bits ℓ_{i+1}. The hashgen algorithm then proceeds by calculating the number of hash calls required to produce the requested amount of output, before hashing the input v_i, incrementing it between hashes and concatenating the output, which is truncated as necessary. It returns the concatenated hashes as output.

hash_drbg.seedgen The seedgen algorithm of the hash_drbg has several different options; for simplicity we will assume that each part of the seed is sampled from random. We will also set $|\mathsf{nonce}| = n$ and both $\mathsf{perstring}, \mathsf{add} \in \{0,1\}^{2^{35}}$.

hash_drbg.setup The setup algorithm of the hash_drbg takes as input an entropy input I_0 output by the entropy source with leakage z_0 and entropy estimate γ_0, along with the seed. The setup algorithm makes a call to the hash_df function, passing the entropy input concatenated with the nonce and personalisation string from the seed. It sets the output of hash_df as the initial v-state, which it then prepends with 0x00 and again passes to hash_df. The output of hash_df becomes the initial c-state. The setup algorithm then sets the reseed counter to 1 and returns the combined values as the initial state of the hash_drbg.

hash_drbg.refresh The refresh algorithm takes as input an entropy input I along with the seed and current state s. The refresh algorithm calls the hash_df algorithm on the current v-state prepended with 0x01, and appended with the entropy input and add of the seed. The result is the updated v-state, which in turn is used via hash_df prepended with 0x00 to produce the updated c-state. The rc is reset back to 1 and these values are output as the refreshed hash_drbg state.

hash_drbg.next The next algorithm takes as input the current state of the generator s_i, along with the seed seed and the requested number of bits $\ell i + 1$. First the current v-state is hashed along with part of the seed and prepended by 0x02. This hashed output is then added to the v_i-state modulo 2^n, since the output of the hash function is shorter than the state length. This updated v-state v_i' is input into the hashgen algorithm along with the requested number of bits ℓ_{i+1}. The hashgen algorithm outputs an updated v-state v_{i+1}' and generator output r_{i+1}. This v-state v_{i+1}' is then hashed, prepending with 0x03 to get u. The next algorithm then calculates $v_{i+1}' + u + c_i + \mathsf{rc}$ modulo 2^n to again attain the correct length. The calculated value is then returned as the updated v-state v_{i+1}, while c_i remains the same and rc is incremented by 1.

hash_drbg.tick The tick algorithm in the hash_drbg is a dummy algorithm that leaves the state unchanged. It is included for completeness of the generator description, as per Definition 4.2.1.

5.3 The ctr_drbg

5.3.1 Notation

Table 5.4 describes the additional notation required for the ctr_drbg. The state of the ctr_drbg is $s := (v, k, \mathsf{rc})$. This differs from the hash_drbg by replacing the k-state with a key state k, which is also designed to retain entropy from the last entropy input and to separate outputs between refreshes.

Our Notation	NIST Notation	Size (bits)	Description
ctr_drbg	CTR_DRBG		The NIST generator based on an approved block cipher.
$\mathsf{E}_k(v)$	Block_Encrypt(key,input_block)	n_E	Block cipher encryption of an input v under a key k.
bcdf	Block_Cipher_df		An auxiliary algorithm.
bcc	bcc		An auxiliary algorithm.
bcup	ctr_drbg _Update		An auxiliary algorithm.
s	seed_material		
n_E	blocklen		Size of the block cipher's input and output block.
n_k	keylen		Key length of the block cipher.
ctrlen	ctr_len	ctrlen	Size of the counter.
ℓ_max (for $B := (2^{\mathsf{ctr\_len}} - 4) * n_\mathsf{E}$)	max_number_of _bits_per_request	$\min(B, 2^{19})$ bits.	Maximum requested number of bits from the generate algorithm.
ℓ_total			Maximum cumulative requested number of bits.
v_i	V	n_E	Sub-state of the working state.
k_i	Key	n_k	Sub-state of the working state.
$s_i := (v_i, k_i, \mathsf{rc})$	Working_State		State of the ctr_drbg.

Table 5.4: Notation for the ctr_drbg, updated for continuity with our notation.

	3 Key TDEA	AES-128	AES-192	AES-256
n_{E}	64	128		
ctrlen	$4 \le \mathsf{ctrlen} \le n_{\mathsf{E}}$			
n_{k}	168	128	192	256
$n = n_{\mathsf{E}} + n_{\mathsf{k}}$	232	256	320	384
If a df is used (not used):				
p	$\lambda \le p \le 2^{35}$ (n)			
|perstring|	$\le 2^{35}$ (n)			
|add|	$\le 2^{35}$ (n)			
ℓ_{max} (for $B := (2^{\mathsf{ctrlen}} - 4) * n_{\mathsf{E}}$)	$\min(B, 2^{13})$	$\min(B, 2^{19})$		
Number of requests between reseeds	2^{32}	$\le 2^{48}$		

Table 5.5: Parameters of the different ctr_drbg instantiations.

5.3.2 Specification of the Generator

The ctr_drbg, like the hash_drbg, has several instantiations (again called "envelopes" in the NIST standard) with associated values given in Table 5.5.

Definition 5.3.1. In addition to the notation given in Table 5.4 and parameters in Table 5.5, let

$$\begin{aligned}
n_{\mathsf{k}} &\approx 2n_{\mathsf{E}} \\
\mathsf{E}_k(v) : \{0,1\}^{n_{\mathsf{k}}} \times \{0,1\}^{n_{\mathsf{E}}} &\longrightarrow \{0,1\}^{n_{\mathsf{E}}}, \\
\lambda \le p &\le 2^{35}, \\
\ell_i \le \ell_{\mathsf{max}} &:= \min(B, 2^{19}), \\
\mathsf{IFace} &:= \{0\}, \\
(\mathsf{nonce}, \mathsf{perstring}, \mathsf{add}) \in \mathsf{Seedspace} &:= \{0,1\}^* \times \{0,1\}^{\le 2^{35}} \times \{0,1\}^{\le 2^{35}}, \\
(v_i, k_i, \mathsf{rc}) \in \mathsf{Statespace} &:= \{0,1\}^{n_{\mathsf{E}}} \times \{0,1\}^{n_{\mathsf{k}}} \times \{0,1\}^{49}.
\end{aligned}$$

Where $\mathsf{E}_k()$ is the chosen block cipher. The ctr_drbg is specified in Figure 5.8, making use of three auxiliary algorithms given in Figure 5.7.

```
$\mathsf{bcup}(x, k, v)$

$y \leftarrow \mathsf{null}$
while $\mathsf{Len}(y) < n$ do
   if $\mathsf{ctrlen} < n_{\mathsf{E}}$
      $\mathsf{inc} \leftarrow (\mathsf{R}_{\mathsf{ctrlen}}(v)) \bmod 2^{\mathsf{ctrlen}}$
      $v \leftarrow \mathsf{L}_{n_{\mathsf{E}}-\mathsf{ctrlen}}(v) \| \mathsf{inc}$
   else
      $v \leftarrow v + 1 \bmod 2^{n_{\mathsf{E}}}$
   $y \leftarrow y \| \mathsf{E}_k(v)$
$y \leftarrow \mathsf{L}_n(y)$
$y \leftarrow y \oplus x$
$k \leftarrow \mathsf{L}_{n_{\mathsf{k}}}(y)$
$v \leftarrow \mathsf{R}_{n_{\mathsf{E}}}(y)$
return $(k, v)$
```

```
$\mathsf{bcc}(k, x)$

$\mathsf{cv} \leftarrow 0^{n_{\mathsf{E}}}$
$m \leftarrow \mathsf{Len}(x)/n_{\mathsf{E}}$
parse $x$ as $\mathsf{block}_1$ to $\mathsf{block}_m$
for $i = 1$ to $m$
   $\mathsf{cv} \leftarrow \mathsf{cv} \oplus \mathsf{block}_i$
   $\mathsf{cv} \leftarrow \mathsf{E}_k(\mathsf{cv})$
return $\mathsf{cv}$
```

```
$\mathsf{bcdf}(x, \ell)$

$L \leftarrow \mathsf{Len}(x)/8$
$N \leftarrow \ell/8$
$x \leftarrow L \| N \| x \| \mathrm{0x80}$
while $(\mathsf{Len}(x) \bmod n_{\mathsf{E}}) \neq 0,$ do
   $x \leftarrow x \| \mathrm{0x00}$
$y \leftarrow \mathsf{null}$
$i \leftarrow 0$
$k \leftarrow \mathsf{L}_{n_{\mathsf{k}}}(\mathrm{0x00010203} \ldots \mathsf{1D1E1F})$
while $\mathsf{Len}(y) < n_{\mathsf{k}} + n_{\mathsf{E}},$ do
   $\mathsf{cv} \leftarrow i \| 0^{n_{\mathsf{E}} - \mathsf{Len}(i)}$
   $y \leftarrow y \| \mathsf{bcc}(k, (\mathsf{cv} \| x))$
   $i \leftarrow i + 1$
$k \leftarrow \mathsf{L}_{n_{\mathsf{k}}}(y)$
$x \leftarrow \mathsf{Sel}^{n}_{n_{\mathsf{k}}+1}(y)$
$y \leftarrow \mathsf{null}$
while $\mathsf{Len}(y) < \ell,$ do
   $x \leftarrow \mathsf{E}_k(x)$
   $y \leftarrow y \| x$
return $\mathsf{L}_{\ell}(y)$
```

Figure 5.7: The auxiliary algorithms bcup, bcdf and bcc of the ctr_drbg.

```
$\underline{\text{ctr\_drbg.next}(\text{seed}, s_i, \ell_{i+1})}$
**parse** seed **as** (nonce, perstring, add)
**parse** $s_i$ **as** $(v_i, k_i, \text{rc})$
**if** add $\neq$ null
    add $\leftarrow$ bcdf(add, $n$)
    $(k, v) \leftarrow$ bcup(add, $k_i, v_i$)
**else**
    add $\leftarrow 0^n$
$y \leftarrow$ null
**while** Len($y$) $< \ell_{i+1}$ **do**
    **if** ctrlen $< n_\mathsf{E}$
        inc $\leftarrow (\mathsf{R}_{\text{ctrlen}}(v) + 1) \bmod 2^{\text{ctrlen}}$
        $v \leftarrow \mathsf{L}_{n_\mathsf{E}-\text{ctrlen}}(v) \| inc$
    **else**
        $v \leftarrow v + 1 \bmod 2^{n_\mathsf{E}}$
    $y \leftarrow y \| \mathsf{E}_k(v)$
$r_{i+1} \leftarrow \mathsf{L}_{\ell_{i+1}}(y)$
$(k_{i+1}, v_{i+1}) \leftarrow$ bcup(add, $k, v$)
rc $\leftarrow$ rc + 1
**return** $((v_{i+1}, k_{i+1}, \text{rc}), r_{i+1})$
```

```
$\underline{\text{ctr\_drbg.setup}(\text{seed}, (\mathrm{I}_0, \gamma_0, z_0))}$
**parse** seed **as** (nonce, perstring, add)
s $\leftarrow$ bcdf($\mathrm{I}_0$‖nonce‖perstring, $n$)
$k \leftarrow 0^{n_\mathsf{k}}$
$v \leftarrow 0^{n_\mathsf{E}}$
$(k_0, v_0) \leftarrow$ bcup(s, $v, k$)
rc $\leftarrow 1$
**return** $s_0 = (v_0, c_0, \text{rc})$
```

```
$\underline{\text{ctr\_drbg.refresh}(\text{seed}, s_i, \mathrm{I})}$
**parse** seed **as** (nonce, perstring, add)
**parse** $s_i$ **as** $(v_i, k_i, \text{rc})$
s $\leftarrow$ I‖add
s $\leftarrow$ bcdf(s, $n$)
$(k_{i+1}, v_{i+1}) \leftarrow$ bcup(s, $k_i, v_i$)
rc $\leftarrow 1$
**return** $(v_{i+1}, k_{i+1}, \text{rc})$
```

```
$\underline{\text{ctr\_drbg.tick}(\text{seed}, s_i)}$
$s_{i+1} \leftarrow s_i$
**return** $s_{i+1}$
```

Figure 5.8: The algorithms describing the behaviour of the ctr_drbg in the format of Definition 4.2.1.

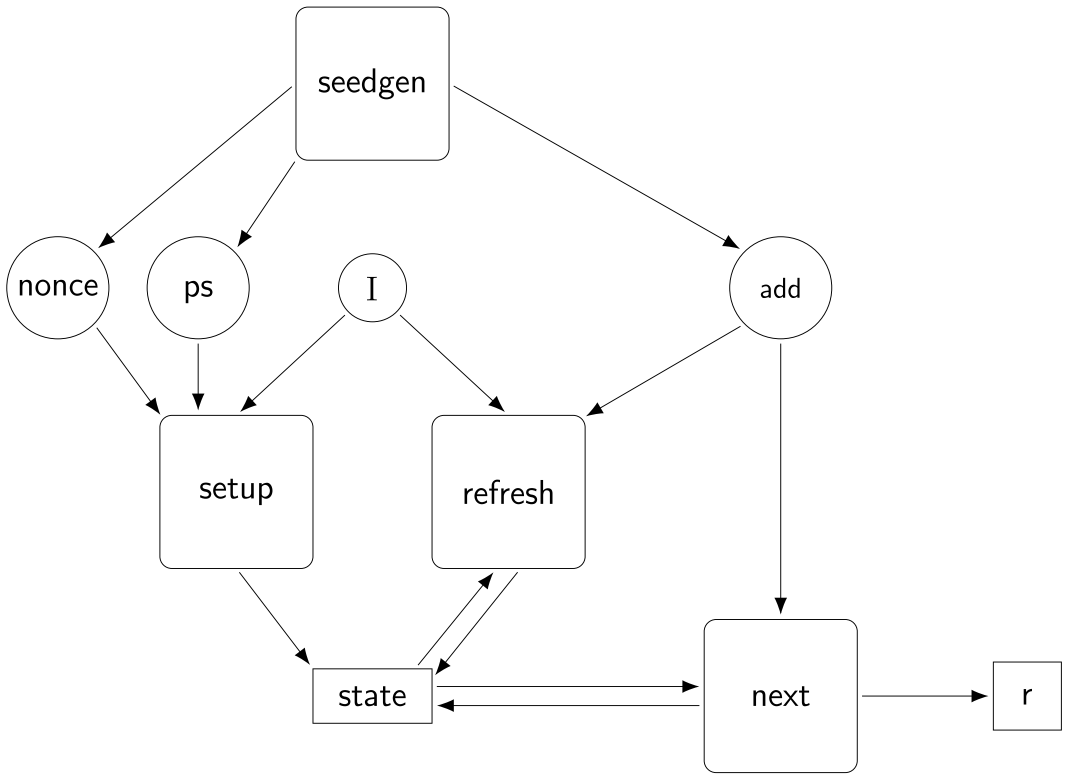


Figure 5.9: The ctr_drbg algorithms and seed usage.

We now provide several overview diagrams of the different ctr_drbg algorithms including their subroutines for clarity.

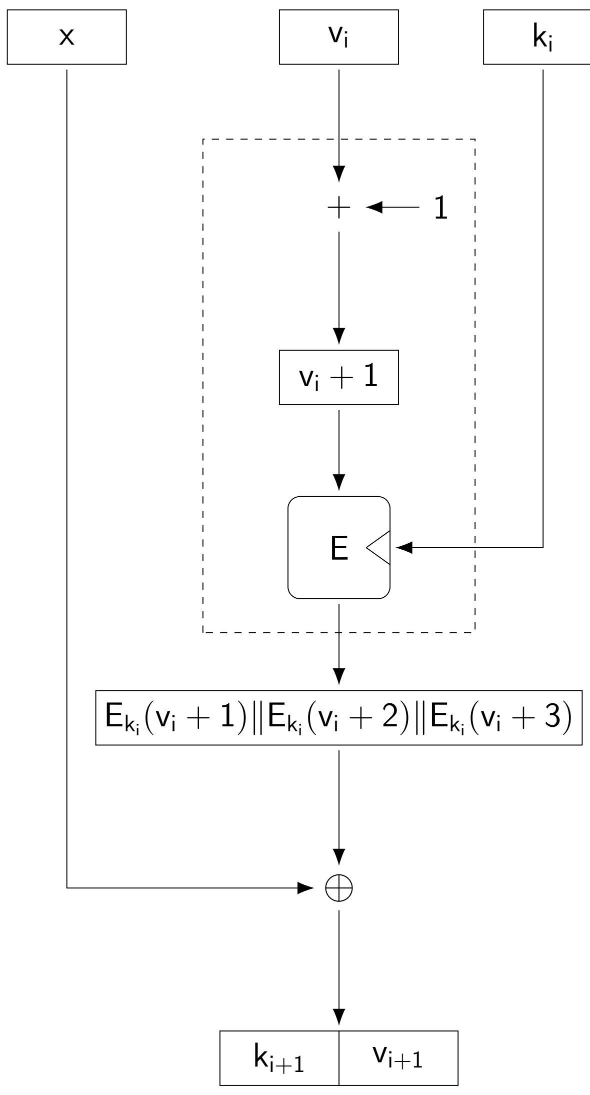


Figure 5.10: The ctr_drbg bcup algorithm.

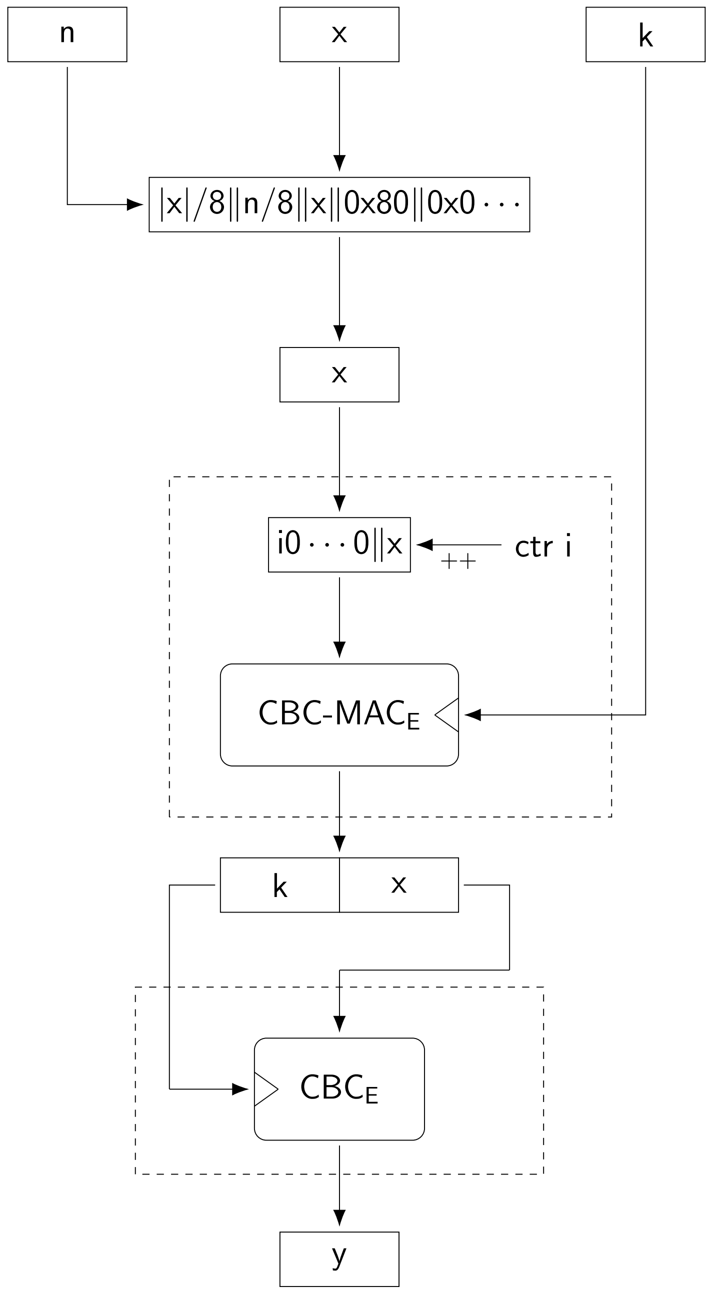


Figure 5.11: The ctr_drbg bcdf algorithm.

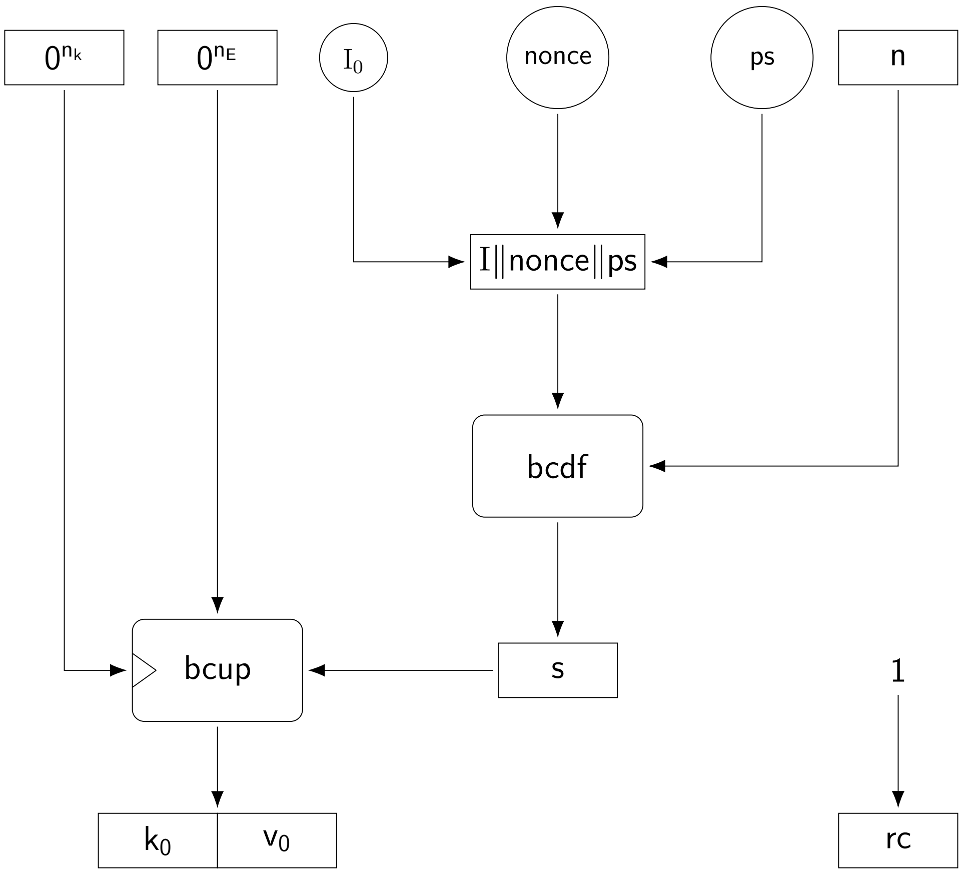


Figure 5.12: The ctr_drbg setup algorithm.

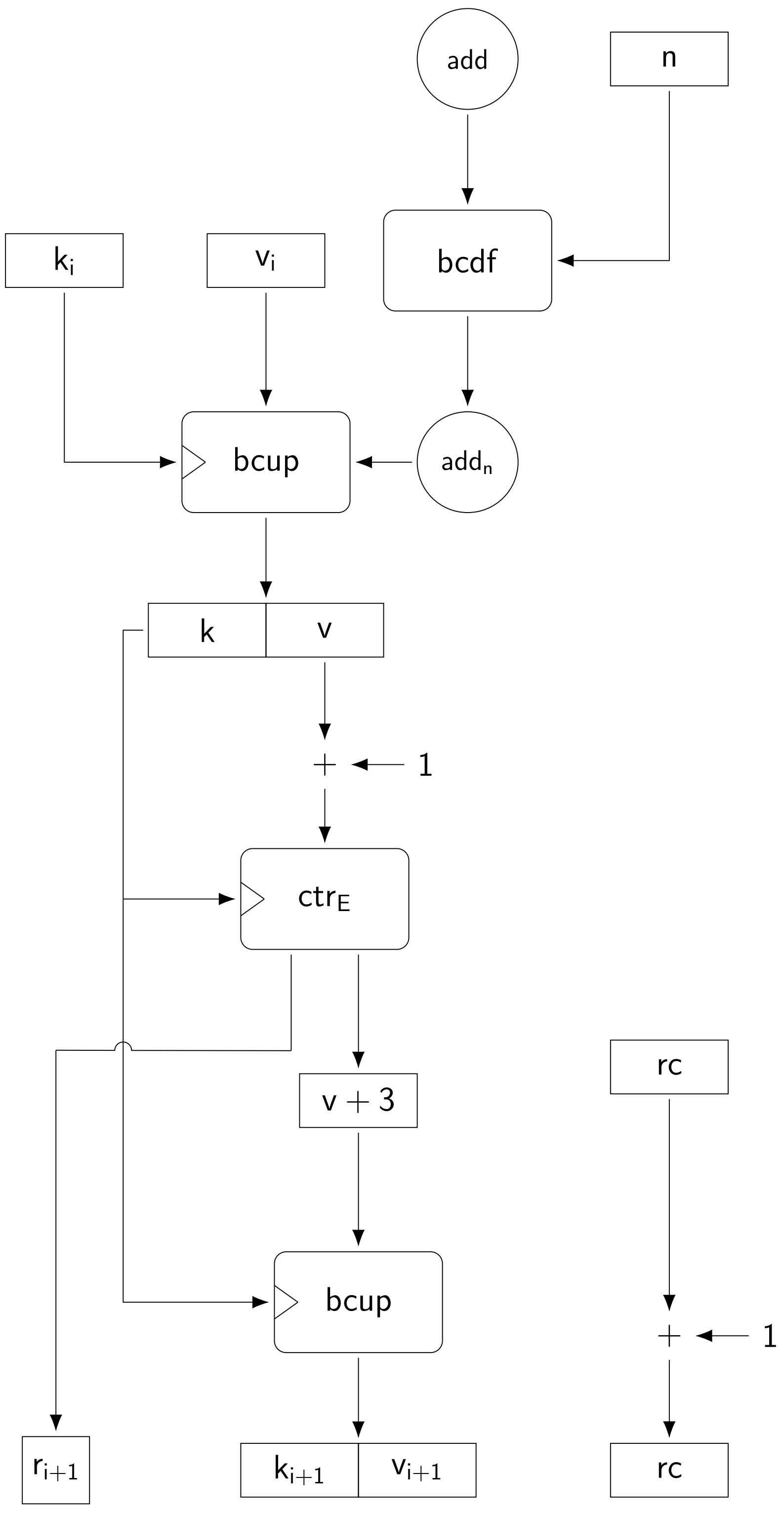


Figure 5.13: The ctr_drbg next algorithm.

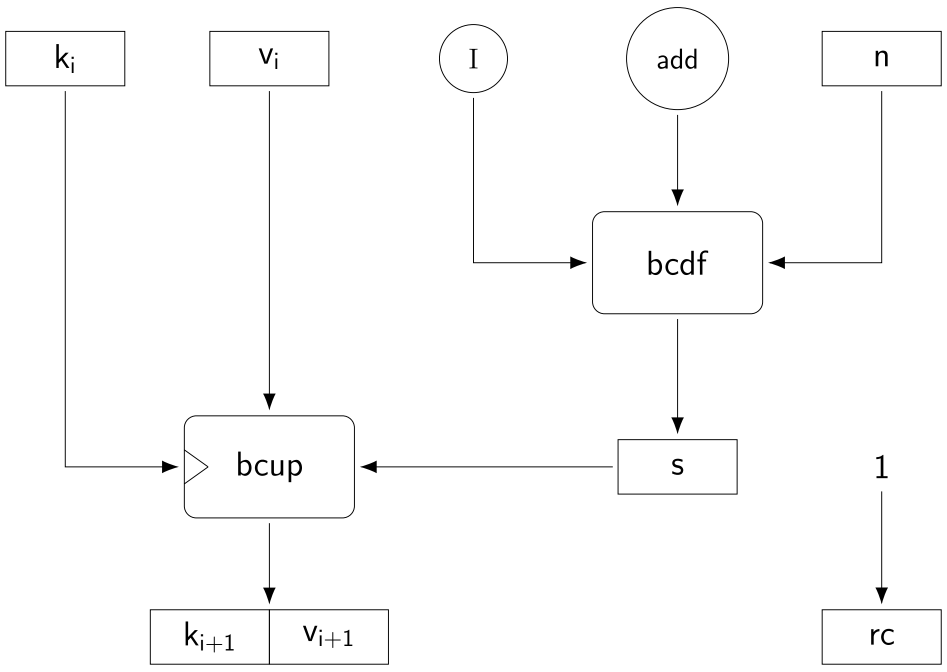


Figure 5.14: The ctr_drbg refresh algorithm.

5.3.3 Algorithm Descriptions

bcup The block cipher update function is an auxiliary algorithm of the ctr_drbg. On input of a string x, key k and state v, repeatedly encrypts v, incrementing the rightmost ctrlen bits of v between encryptions. This is done until the concatenated output is n-bits long (after truncation if necessary), which is then XORed with the input x and split into the new k and v as leftmost and rightmost parts of the output respectively. This is used to refresh the state with fresh entropy or update it between calls to next.

bcdf The block cipher derivation function is the second auxiliary algorithm of the ctr_drbg. For a given input string x and requested number of output bits ℓ, the bcdf algorithm calculates the number of bytes of both input and output requested. These values are then concatenated with the input x, which is appended with 0x80 and padded until it is a multiple of the n_{E}. Next, the bcdf algorithm calls the bcc algorithm using a set key, a counter and the input x. The output of the bcc algorithm is then stored as a temporary string, the counter is incremented and the bcc algorithm is called again in the same way with the output appended to the temporary string. This is repeated until the temporary string is of a length greater than or equal to $n = n_{\mathsf{k}} + n_{\mathsf{E}}$. This concatenated output is then truncated to n-bits before being split into a new key k and updated x. These updated values are then used as inputs to the block cipher which is called a number of times, with each output being the next input to the block cipher while concatenating the output in a temporary string. Finally, the concatenated output of the block cipher is truncated to the requested number of bits ℓ and returned as output of the algorithm. The algorithm is designed to derive internal state or distribute entropy throughout a bitstring.

This algorithm essentially runs CBC-MAC with three iterations (since $n \approx 3n_{\mathsf{E}}$), with a counter separating the inputs to produce an intermediate (k, v), which are then used in CBC mode to generate the required amount of output (e.g. three blocks to be used as the updated state).

bcc The bcc algorithm is the third auxiliary algorithm of the ctr_drbg which takes as input a key k and input string v. It initialises a chaining value to 0, splits the input v into blocks of length n_{E} and updates the chaining value by XORing it with the first block. The algorithm then encrypts the chaining value under the given key and repeats this process until all blocks have been incorporated. The bcc algorithm then returns the final chaining value as output.

ctr_drbg.seedgen The seedgen algorithm of the ctr_drbg has several different options; for simplicity we will assume that each part of the seed is sampled from random. We will also set $|\mathsf{nonce}| = n$ and both $\mathsf{perstring}, \mathsf{add} \in \{0,1\}^{2^{35}}$.

ctr_drbg.setup The setup algorithm of the ctr_drbg requires an entropy input I_0. The entropy input I_0 is output by the entropy source, together with leakage z_0, and entropy estimate γ_0. The setup algorithm also requires the seed. The setup algorithm makes a call to the bcdf algorithm, passing the entropy input, concatenated with the nonce nonce and personlisation string perstring of the seed. It then initialises the k-state k and v-state v as zero strings, inputs these and the previous output of the bcdf algorithm into the bcup algorithm, which outputs values for the initial state values k_0 and v_0. It sets the reseed counter to 1 and outputs the combined values as the initial state of the ctr_drbg.

ctr_drbg.refresh The refresh algorithm takes as input an entropy input I along with the seed and current state s of the ctr_drbg. The algorithm calls the bcdf algorithm on I and additional input add of the seed. The output of the bcdf algorithm is then input, along with the current k-state k and v-state v into the bcup algorithm, which returns updated k and v-states. The reseed counter is reset back to 1 and the updated values (v, k, rc) are returned as the new ctr_drbg state.

ctr_drbg.next The next algorithm takes the current state of the generator s_i as input, along with the seed and the requested number of bits ℓ_{i+1}. The next algorithm first updates the current k-state k_i and v-state v_i using the additional input add of the seed (which may be the zero string) by inputting these values into the bcup algorithm. The updated values k_i' and v_i' are used as input to the block cipher, with output written to a temporary string. The value of v_i' is incremented and encrypted again under k_i' with the output appended to the temporary string. This is repeated until the temporary string is at least $\ell i + 1$. The temporary string is truncated to ℓ_{i+1} and returned as the generator output. The next algorithm then inputs the current values k_i', v_i' into the bcup algorithm, along with add. The output of the bcup algorithm is returned as the updated values k_{i+1} and v_{i+1} along with the generator output r_{i+1}.

ctr_drbg.tick The tick algorithm in the ctr_drbg is a dummy algorithm that leaves the state unchanged. It is included for completeness of the generator by Definition 4.1.2.

5.4 Security of the hash_drbg

We begin by noting that since the c-state of the generator depends on the last refreshed v-state, the adversary can easily distinguish between s and $\mathsf{M}(s)$ after a refresh by checking $\mathsf{hash\_df}((0\text{x}00\|v), n) == c$. This motivates the split masking function to treat a refresh or initialisation masking differently than after a call to next.

Below we define the split masking function M of the hash_drbg. We note that the c depends entirely on the initial v after a refresh or initialise, but remains constant otherwise.

5.4.1 Masking Function of hash_drbg

Definition 5.4.1. For Statespace defined as in Definition 5.2.1, the hash_drbg masking function, is as follows:

$$\begin{aligned}
\mathsf{M} &:= (\mathsf{M_I}, \mathsf{M_P}, \mathsf{M_R}) : \{0,1\}^{2n+49} \longrightarrow \{0,1\}^{2n+49},\\
\mathsf{M_I}(s) &= (v', c', 1) \quad \text{where } v' \xleftarrow{\$} \{0,1\}^n, c' = \mathsf{hash\_df}((0\text{x}00 \| v'), n),\\
\mathsf{M_P}(s) &= (v', c, \mathsf{rc}) \quad \text{where } v' \xleftarrow{\$} \{0,1\}^n,\\
\mathsf{M_R}(s) &= (v', c', \mathsf{rc}) \quad \text{where } v', c' \xleftarrow{\$} \{0,1\}^n.
\end{aligned}$$

This captures how the notion of the "ideal state" of the generator changes depending on the situation.

- After initialisation we expect a random v-state, but the c-state should be entirely dependent on the current v-state.
- After a preserving next call we expect the v-state to still be random, but the c-state is expected to remain constant, being dependent on the v-state closest to the last entropy input.
- Lastly, after a recovering next call we expect the v-state to again be random, but we also expect the c-state to have been updated, but not directly dependent on the *current* v-state since it will have been updated as part of the next call.

Lemma 5.4.2.
The hash_drbg split masking function as defined in Definition 5.4.1 is idempotent.

Proof. For a state $s \in$ Seedspace, the split masking function of the hash_drbg always fixes the "public" part the reseed counter rc and overwrites the v-state with uniformly random bits. We look at the separate cases, focusing on the first mask applied:

- If the first mask is $\mathsf{M_I}$ then s must be a result of setup and will be of the form $(v, c := f(v), \mathsf{rc})$ where $f(x)$ is $\mathsf{hash\_df}((0\text{x}00\|x), n)$. We note that $\mathsf{M_I}(s)$ is identically distributed to this s with a random v-state. With this knowledge, applying either $\mathsf{M_P}$ or $\mathsf{M_R}$ will always overwrite the v-state, leaving $\mathsf{rc} = 1$. In the case of $\mathsf{M_R}$, the c-state will also be overwritten, so the resulting state will always be of the form $(v', c', 1)$ for v', c' chosen uniformly from random. In the case of $\mathsf{M_P}$, the v will always be chosen uniformly from random, while the c is a function of the previous v and $\mathsf{rc} = 1$. In both cases, the mask of the masked state is identically distributed to the masked state.

- If the first mask is $\mathsf{M_P}$ then only the v-state will have been overwritten as uniformly random. Applying either $\mathsf{M_P}$ or $\mathsf{M_R}$ afterwards will overwrite at least the v-state, leaving the rcunchanged. Depending on the choice of second mask the c-state is either overwritten or left constant, in either case the state is left identically distributed to $\mathsf{M_P}(s)$.

- If the first mask is $\mathsf{M_R}$ then the v-state and c are overwritten. If the second mask is also $\mathsf{M_R}$ then the v-state and c-state are again overwritten, making the double masking identically distributed to $\mathsf{M_R}(s)$. If the second mask is $\mathsf{M_P}$ then the v-state is overwritten, the c is left constant, which is still uniformly random, and rcis left constant. This makes the double masking also identically distributed to $\mathsf{M_R}(s)$.

□

Lemma 5.4.3.
The split masking function of the hash_drbg defined in Definition 5.4.1 is a $(\mathcal{D}, t, \epsilon_{\mathsf{h}})$-honest initialisation split masking function under the assumption that the entropy source $\mathcal{D}$ outputs at minimum k bits of entropy and where $\epsilon_{\mathsf{h}} := \frac{1}{2}\sqrt{2^{2n_{\mathsf{H}}-k} + \alpha(\alpha+2)}$.

Proof. Since $\mathsf{M_I}$ overwrites the v-state with uniformly random bits, while calculating the resulting c-state in the same way as the setup algorithm, an adversary distinguishing between s_0 and $\mathsf{M_I}(s_0)$ would be able to distinguish the output of the hash_df algorithm from random. We will therefore show that the hash_df algorithm is a $(k, \epsilon_{\mathsf{h}})$-extractor and thus

$$\mathsf{Adv}^{\mathsf{init}}_{\mathsf{G},\mathcal{D},\mathsf{M}}(\mathcal{A}) \leq \epsilon_{\mathsf{h}}.$$

Before we discuss details of the hash_df function, we look at the advice given in the NIST document about underlying hash functions. We note that all of the combinations given mean that at most, hash_df in the initialise algorithm would call the underlying hash function thrice; this stems from the required length n being at most three times the size of n_{H}. We will in fact restrict ourselves to the situation with SHA256 and SHA512 for simplicity, both of which only require two calls.

Lemma 5.4.4.
In the context of the hash_drbg.setup algorithm, let $\mathsf{H} : \{0,1\}^* \longrightarrow \{0,1\}^{n_{\mathsf{H}}}$ *be a strong* ρ*-universal hash function, with the family defined over* nonce *and* perstring*, which we assume to be uniformly random strings. Let* $\rho := (1+\alpha)2^{-n_{\mathsf{H}}}$*. Then the hash_df function is a* ρ'*-universal hash function* $\mathsf{H}' : \{0,1\}^* \longrightarrow \{0,1\}^{2n_{\mathsf{H}}}$ *and again with the family defined over* nonce *and* perstring*, with* $\rho' := (1+\alpha')2^{-2n_{\mathsf{H}}}$*. This yields* $\alpha' = \alpha(\alpha+2)$*.*

Proof. Assuming that the hash_df outputs the entire output string, we have that $\mathsf{H}'(x) := \mathsf{H}(1\|\ell\|x)\|\mathsf{H}(2\|\ell\|x)$, where ℓ is the size of the output, and thus a constant. Since the counter separates the two inputs to H we have that for all $x_1, x_2 \in \{0,1\}^n$, $x_1 \neq x_2$, for a collision to occur, both $\mathsf{H}(1\|\ell\|x_1) = \mathsf{H}(1\|\ell\|x_2)$ and $\mathsf{H}(2\|\ell\|x_1) = \mathsf{H}(2\|\ell\|x_2)$, which are both bounded as at most ρ. Thus a collision will occur with probability ρ^2.

Defining $\mathsf{H}' : \{0,1\}^n \longrightarrow \{0,1\}^{2n_{\mathsf{H}}}$, we can conclude that H' is a strong $\rho' := \rho^2$-universal hash function. This yields $\rho' = (1+\alpha')2^{-2n_{\mathsf{H}}} = \rho^2 = (1+\alpha)^2 2^{-2n_{\mathsf{H}}} \implies \alpha' = \alpha(\alpha+2)$. □

Lemma 5.4.5.
In the context of the hash_drbg.setup algorithm, assuming the underlying hash function $\mathsf{H} : \{0,1\}^* \longrightarrow \{0,1\}^{n_{\mathsf{H}}}$ *is a strong* ρ*-universal hash function family over the space of all* nonce *and* perstring *(*$\{0,1\}^d$ *where* $d \leq |\mathsf{nonce}| + 2^{35}$*). For an entropy source with minimum entropy* k *and with entropy loss* $L := k - 2n_{\mathsf{H}}$*, by Lemma 5.4.4 and by applying the leftover hash lemma (Lemma 2.4.5), we have that the hash_df algorithm is a* (k,ϵ)*-extractor for* $\epsilon := \frac{1}{2}\sqrt{2^{2n_{\mathsf{H}}-k} + \alpha(\alpha+2)}$*.*

Proof. By Lemma 5.4.4 we have that hash_df is a ρ'-universal hash function family over the same space. We then use the leftover hash lemma that states that hash_df, as described in Figure 5.1, is a (k, ϵ)-extractor where $\epsilon := \frac{1}{2}\sqrt{2^{2n_{\mathsf{H}}-k} + \alpha'}$. We also need to assert that the seed length meets the required length; namely, $d \geq \min(|\mathrm{I}| - 2n_{\mathsf{H}}, 2n_{\mathsf{H}} + 2\log(1/\epsilon) - O(1)) = \min(p - 2n_{\mathsf{H}}, 2n_{\mathsf{H}} + 2\log(1/\epsilon) - O(1))$. Since the seed length is $d = |nonce| + 2^{35}$, this is easily met. $\square$

Lemma 5.4.5 also concludes the proof of Lemma 5.4.3. $\square$

This requires several fairly strong assumptions that do not translate to a standard implementation of the hash_drbg; namely, the nonce and perstring are not generally uniformly random, and the underlying hash function, e.g. SHA512, will yield a ρ resulting in large α. For now, we will proceed with our assumptions to prove our security claims.

Corollary 5.4.6.
In the context of the hash_drbg.refresh algorithm, assuming that the add is chosen uniformly at random, the hash_df is a $(k, \frac{1}{2}\sqrt{2^{2n_{\mathsf{H}}-k} + \alpha^*(\alpha^*+1)})$-extractor for the above parameters when used to refresh the v state. Similarly the seed length for the extractor is 2^{35} bits which meets the required minimum length.

Proof. Following the same argument as Lemma 5.4.4, we obtain that the hash_df is a ρ''-universal hash function in the same way but with the family over $\{0,1\}^{\mathsf{add}}$, hence we use α^*. $\square$

5.4.2 PRG Security of the next Function of the hash_drbg

We now need to consider the PRG security of the next function. We could model it similarly to [34], where the circuits are describing the counter used in the generate algorithm, but since the adversary obtains all of the output and the circuits are so restricted, it makes more sense to approach the proof more directly.

We do this by defining the security game res_1 given in Figure 5.15.

```
$\mathsf{res_1}(\mathcal{A}_{\mathsf{PRG}}, \mathsf{D_i}, \mathsf{D_{i+1}}, \ell_{i+1})$
$\mathsf{seed} \leftarrow \mathsf{seedgen}; \mathsf{b} \overset{\$}{\leftarrow} \{0,1\}$
$s_0' \overset{\$}{\leftarrow} \{0,1\}^{2n+49}$
$s_0 \leftarrow \mathsf{M_I}(s_0')$
$(s_{i+1}, r_{i+1}) \leftarrow \mathsf{D_{i+b}}(\mathsf{seed}, \mathsf{M}(s_0'), \ell_{i+1})$
$\mathsf{b}^* \overset{\$}{\leftarrow} \mathcal{A}_{\mathsf{next}}((s_{i+1}, r_{i+1}), \mathsf{rc})$
return $\mathsf{b} == \mathsf{b}^*$
```

Figure 5.15: The security game for distinguishing between hash_drbg.next and a random generator output and masked state.

Definition 5.4.7. For an algorithm $\mathsf{D_i}$ representing the next function, for a masked input state $\mathsf{M_I}(s_0)$, seed seed, requested number of bits $\ell_{i+1} \leq \ell_{\mathsf{max}}$, and any adversary $\mathcal{A}_{\mathsf{PRG}}$ running in time no more than t playing the game described in Figure 5.15, which is derived from the resilence notion (from Definition 4.4.2) restricted to one call to next, has advantage

$$\mathsf{Adv}^{\mathsf{res_1}}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D_i}(s_i, \ell_{i+1}), \mathsf{D_{i+1}}(s_i, \ell_{i+1})) \leq \left|\Pr\left[\mathcal{A}^{\mathsf{D_i}}_{\mathsf{PRG}} = 1\right] - \Pr\left[\mathcal{A}^{\mathsf{D_{i+1}}}_{\mathsf{PRG}} = 1\right]\right|.$$

Lemma 5.4.8.
Let s_i be a random variable on the Statespace*, then for any q adversary $\mathcal{A}_{\mathsf{PRG}}$, and $\ell_{i+1} \leq \ell_{max}$,*

$$\mathsf{Adv}^{\mathsf{res_1}}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{next}(s_i, \ell_{i+1}), (\mathsf{M}(s_{i+1}), \mathsf{U}_{\ell_{i+1}})) \leq \epsilon_{\mathsf{PRG}} \leq \left(\left\lceil \frac{\ell_{i+1}}{n_{\mathsf{H}}} \right\rceil + 2\right) \mathsf{Adv}^{\mathsf{owf}}_{\mathsf{H}}(q, n_{\mathsf{H}}).$$

Proof. The proof will proceed in two parts. First we will show that an adversary cannot distinguish between the real or masked output state, while always having real output. Following this, given masked output state the adversary cannot distinguish between real and random output. Recall $m := \lceil \ell_{i+1}/n_{\mathsf{E}} \rceil$.

We define three algorithms in Figure 5.16 that we will use for the hybrid game jumps. We omit parsing the seed for simplicity.

```
Alg. $\mathsf{D}_0(\mathsf{seed}, s_i, \ell_{i+1})$
    parse $s_i$ as $(v_i, c_i, \mathsf{rc})$
    $w \leftarrow \mathsf{H}(0x02 \| v_i \| \mathsf{add})$
    $v \leftarrow (v_i + w) \mod 2^n$
    $(r_{i+1}, v) \leftarrow \mathsf{hashgen}(\ell_{i+1}, v)$
    $u \leftarrow \mathsf{H}(0x03 \| v)$
    $v_{i+1} \leftarrow (v + u + c_i) \mod 2^n$
    $c_{i+1} \leftarrow c_i$
    $\mathsf{rc} \leftarrow \mathsf{rc} + 1.$
    return $(r_{i+1}, s_{i+1})$

Alg. $\mathsf{D}_1(\mathsf{seed}, s_i, \ell_{i+1})$
    parse $s_i$ as $(v_i, c_i, \mathsf{rc})$
    $w \leftarrow \mathsf{H}(0x02 \| v_i \| \mathsf{add})$
    $v \leftarrow (v_i + w) \mod 2^n$
    $(r_{i+1}, v) \leftarrow \mathsf{hashgen}(\ell_{i+1}, v)$
    $u \leftarrow \mathsf{H}(0x03 \| v)$
    $v_{i+1} \leftarrow (v + u + c_i) \mod 2^n$
    $c_{i+1} \leftarrow c_i$
    $\mathsf{rc} \leftarrow \mathsf{rc} + 1.$
    $s_{i+1} \leftarrow \mathsf{M}(s)$
    return $(r_{i+1}, s_{i+1})$

Alg. $\mathsf{D}_2(\mathsf{seed}, s_i, \ell_{i+1})$
    parse $s_i$ as $(v_i, c_i, \mathsf{rc})$
    $w \leftarrow \mathsf{H}(0x02 \| v_i \| \mathsf{add})$
    $v \leftarrow (v_i + w) \mod 2^n$
    $(r_{i+1}, v) \leftarrow \mathsf{hashgen}(\ell_{i+1}, v)$
    $r_{i+1} \xleftarrow{\$} \{0,1\}^{\ell_{i+1}}$
    $u \leftarrow \mathsf{H}(0x03 \| v)$
    $v_{i+1} \leftarrow (v + u + c_i) \mod 2^n$
    $c_{i+1} \leftarrow c_i$
    $\mathsf{rc} \leftarrow \mathsf{rc} + 1.$
    $s_{i+1} \leftarrow \mathsf{M}(s)$
    return $(r_{i+1}, s_{i+1})$
```

Figure 5.16: The algorithms $\mathsf{D}_0, \mathsf{D}_1$ and D_2 used in proving the PRG security of the hash_drbg.next function.

We then have,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{next}(s_i, \ell_{i+1}), (\mathsf{M}(s_{i+1}), \mathsf{U}_{\ell_{i+1}})) &= \mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_0(s_i, \ell_{i+1}), \mathsf{D}_2(s_i, \ell_{i+1}))\\ &\leq \mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_0(s_i, \ell_{i+1}), \mathsf{D}_1(s_i, \ell_{i+1}))\\ &+ \mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_1(s_i, \ell_{i+1}), \mathsf{D}_2(s_i, \ell_{i+1})).\end{aligned}$$

We note that the first part is relatively straightforward. We present it in Lemma 5.4.9.

Lemma 5.4.9.
Let s_i be as described in Definition 5.4.7. Let H be a strong ρ-universal hash function. Then for any adversary working in time t=q, $\mathcal{A}_{\mathsf{PRG}}$, and $\ell_{i+1} \leq \ell_{max}$,

$$\mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_0(s_i, \ell_{i+1}), \mathsf{D}_1(s_i, \ell_{i+1})) \leq 2\mathsf{Adv}^{\mathsf{owf}}_{\mathsf{H}}(q, n_{\mathsf{H}}).$$

Proof. We simplify the distinguishing game by giving the adversary the intermediate values $w \leftarrow \mathsf{H}(0x02 \| v_i \| \mathsf{add})$ and $u \leftarrow \mathsf{H}(0x03 \| v+m)$, once all queries to H have been made. The latter value can be reconstructed using the challenge and the adversary can always disregard the additional information. Therefore,

$$\mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_0(s_i, \ell_{i+1}), \mathsf{D}_1(s_i, \ell_{i+1})) \leq \mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_0(s_i, \ell_{i+1}, w, u), \mathsf{D}_1(s_i, \ell_{i+1}, w, u)).$$

After receiving this information, an adversary must either have found a pre-image of $\mathsf{H}(0\text{x}02\|v_i\|\mathsf{add})$ or found a pre-image of $\mathsf{H}(0\text{x}03\|v+m)$, which by the properties of H is precisely $\mathsf{Adv}^{\mathsf{owf}}_{\mathsf{H}}(q, n_{\mathsf{H}})$ respectively. □

Lemma 5.4.10.
Let s_i be as described in Definition 5.4.7 and let H be a strong ρ-universal hash function. Then for any adversary working in time t=q, $\mathcal{A}_{\mathsf{PRG}}$, and $\ell_{i+1} \leq \ell_{max}$,

$$\mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_1(s_i, \ell_{i+1}), \mathsf{D}_2(s_i, \ell_{i+1})) \leq m\mathsf{Adv}^{\mathsf{owf}}_{\mathsf{H}}(q, n_{\mathsf{H}})\,.$$

Proof. The output the adversary receives is either random, or, of the form $\mathsf{H}(v)\|\mathsf{H}(v+1)\|\cdots\|\mathsf{H}(v+m-1)$, where $m := \ell_{i+1}/n_{\mathsf{H}}$. Similar to Lemma 5.4.9, if we give the adversary the intermediate value $v_i + \mathsf{H}(0\text{x}2\|v_i\|\mathsf{add})$, to succeed, an adversary must have found a pre-image of one of the hash blocks. We also note that each query the adversary makes to H reduces the number of possible pre-images by m, since the output is a chain of values.

This yields adversarial advantage $m\mathsf{Adv}^{\mathsf{owf}}_{\mathsf{H}}(q, n_{\mathsf{H}})$. □

Putting together Lemmas 5.4.9 and 5.4.10 concludes the proof. □

5.4.3 Variable-Output Robustness of the hash_drbg

Theorem 5.4.11.
The hash_drbg is $(t', \ell_{max}, \ell_{total}, \mathsf{q}_{\mathcal{D}}, \mathsf{q}_{\mathsf{S}}, \gamma^*, \epsilon_{\mathsf{rob}})$-variable outupt robust witnessed by the $(\mathcal{D}, t, \epsilon_{\mathsf{h}})$-honest idempotent split masking function M, where

$$\epsilon_{\mathsf{rob}} \leq \epsilon_{\mathsf{h}} + \max_{\substack{(\ell_1,\ldots,\ell_{\mathsf{q}}):\\ \sum_{i=1}^{\mathsf{q}} \ell_i \leq \ell_{total}}} \left(\sum_{i=1}^{q} (\epsilon_{\mathsf{p}} + \epsilon_{\mathsf{r}}) \right),$$

and for legitimate distribution sampler $\mathcal{D}$, under the assumption that H is a ρ-universal hash function for $\rho = (1+\alpha)2^{-n_{\mathsf{H}}}, k = \gamma^* + 2n_{\mathsf{H}}$ and for ϵ_{p} and ϵ_{r} given in the following lemmas.

Proof.

Proof outline The proof proceeds as normal by proving the hash_drbg satisfies both variable output preserving and variable output recovering security.

Lemma 5.4.12.
Let the hash_drbg be as described in Definition 5.2.1. Then the hash_drbg has $(t, \ell_{max}, \epsilon_{\mathsf{p}})$-variable output preserving security, witnessed by the masking function M defined in Definition 5.4.1 and for:

$$\epsilon_{\mathsf{p}} \leq \left(\left\lceil \frac{\ell_{i+1}}{n_{\mathsf{H}}} \right\rceil + 6d + 2 \right) \mathsf{Adv}_{\mathsf{H}}^{\mathsf{owf}}(q, n_{\mathsf{H}}).$$

Proof. The proof proceeds by hybrid argument. First we show that we can replace the intermediate state s_d after the adversarial refreshes have been made with a masked state, followed by using the earlier result of Lemma 5.4.8 to show that the final challenge is indistinguishable from random output and a masked state.

If we give the adversary additional information after she has made all of her queries to H, namely the value of v_{i+1} and the value s, then an adversary able to distinguish between the challenge and a masked state and output must have queried one of these two two values with the correct prefix. Since the adversary is allowed to make d

adversarial refresh calls, this becomes

$$6d\mathsf{Adv}^{\mathsf{owf}}_{\mathsf{H}}\left(q, n_{\mathsf{H}}\right).$$

Putting this together with Lemma 5.4.8 yields

$$\epsilon_{\mathsf{p}} \leq \left(\left\lceil \frac{\ell_{i+1}}{n_{\mathsf{H}}} \right\rceil + 6d + 2\right) \mathsf{Adv}^{\mathsf{owf}}_{\mathsf{H}}\left(q, n_{\mathsf{H}}\right).$$

We set **Game 0** as the VOPWI preserving security game witnessed by the masking function M, **Game 1** is identical to **Game 0** but replacing the intermediate value s_d with a masked state, and finally, **Game 2** replaces the final output with a masked state using Lemma 5.4.8.

Lemma 5.4.13.
Let H *be a ρ-universal hash function for $\rho := (1+\alpha)2^{-n_{\mathsf{H}}}$ and let the* hash_drbg *be as described in Definition 5.2.1. The* hash_drbg *has $(t, \mathsf{q}_{\mathcal{D}}, \ell_{max}, \gamma^*, \epsilon_{\mathsf{r}})$-variable output recovering security, witnessed by the masking function* M *defined in Definition 5.4.1 and for:*

$$\epsilon_{\mathsf{r}} \leq \epsilon^{\mathsf{refresh}'}_{\mathsf{ext}} + \epsilon_{\mathsf{PRG}}.$$

Proof. The proof proceeds with a hybrid argument. First we show that we can replace the intermediate state after the refresh queries with a masked state, followed by using the earlier result of Lemma 5.4.8 to show that the final challenge is indistinguishable from random output and a masked state.

If we modify the refresh algorithm to take as input the entire string of refresh values as an on-line extractor, call it $\mathsf{refresh}'(\mathsf{seed}, s_i, \mathrm{I}')$ for $\mathrm{I} := \mathrm{I}_1, \ldots, \mathrm{I}_d$, we can prove that this new algorithm is a universal hash function and employ the leftover hash lemma.

Lemma 5.4.14.
The algorithm refresh*', as described in the previous paragraph, assuming that* H *is a strong ρ-universal hash function and:*

$$\rho' := \sum_{i=2}^{d} \left(\Pr\left[\mathsf{coll}|\mathbf{Ev_i}\right]\right) = \frac{1}{2^{3n_{\mathsf{E}}}}\left(\rho^*\right),$$

is a ρ'-universal hash function.

Proof. Let $\mathrm{I}'_0 \neq \mathrm{I}'_1$, with $\mathrm{I}'_j := (\mathrm{I}_{j;1} \ldots, \mathrm{I}_{j;d})$. We split into several cases. Let $\mathbf{Ev_i}$ be the event corresponding to I'_0, and I'_1 only differ in the first i blocks, with $\mathbf{Ev_1}$ meaning they differ in the first block alone, while $\mathbf{Ev_d}$ mean they differ in all d blocks. Since the cs are not used to propagate the state, we need only focus on finding a collision in the v.

In $\mathbf{Ev_1}$ the probability of a collision in the first iteration is precisely the probability that there is a collision in v_{i+1} over the choice of add. Since H is a ρ-universal hash function, this probability is precisely $\frac{1}{\rho^3}$. However, even if a collision does not occur in the first block, there is still a non-zero probability that for each block the v_{i+1} will collide, again with probability $\frac{1}{\rho^3}$. Putting this together yields $\Pr[\mathsf{coll}|\mathbf{Ev_i}] = \frac{d}{\rho^3}$, since once a collision has occurred, the remaining blocks will be equal.

Now that we have dealt with the case $\mathbf{Ev_1}$ we can look to the general case of $\mathbf{Ev_i}$. The general case can be split as follows:

- On input block $k \leq i$ there is a collision denoted $\mathsf{coll}(k)$, then there must also be a collision in the remaining $i-k$ blocks that differ between I'_0 and I'_1.
- On input block $k > i$ there is a collision in one of the remaining $(d-k)$ blocks where I'_0 and I'_1 match.

The first case can be calculated to be:

$$\Pr[\mathsf{coll}(k)|\mathbf{Ev_i} \wedge (k \leq i)] = \frac{1}{\rho^3}\frac{1}{(\rho^3)^{i-k}} = \frac{1}{\rho^{3(1+i-k)}}.$$

Which means that for all such k we have:

$$\begin{aligned}\Pr[\mathsf{coll}|\mathbf{Ev_i} \wedge (k \leq i)] &= \frac{1}{\rho^3}\prod_{k=1}^{i-1}\frac{1}{(\rho^3)^{i-k}} = \frac{1}{\rho^3}\prod_{j=1}^{i-1}\frac{1}{(\rho^3)^j} \\ &= \frac{1}{\rho^3}\frac{1}{(\rho^3)^{\frac{i(i-1)}{2}}} = \rho^{-\frac{3(i(i-1)+2)}{2}}. \end{aligned} \tag{5.4.1}$$

The second case is simpler, as follows:

$$\Pr[\mathsf{coll}(k)|\mathbf{Ev_i} \wedge (k > i)] = \frac{1}{\rho^3}.$$

So for all such k this becomes:

$$\Pr[\mathsf{coll}|\mathbf{Ev_i} \wedge (k > i)] = \sum_{k=i+1}^{d}\frac{1}{\rho^3} = \frac{d-i}{\rho^3}. \tag{5.4.2}$$

Putting Equations (5.4.1) and (5.4.2) together yields

$$\begin{aligned}
\rho' := & \sum_{i=2}^{d} \left(\Pr\left[\mathsf{coll}|\mathbf{Ev_i}\right]\right) + \frac{d}{\rho^3} \\
= & \sum_{i=2}^{d} \left(\frac{1}{(\rho^3)^{\frac{i(i-1)+2}{2}}} + \frac{d-i}{\rho^3} \right) + \frac{d}{\rho^3} \\
= & \frac{2d + (d-1)(d-2)}{2\rho^3} + \sum_{i=2}^{d} \left(\frac{1}{(\rho^3)^{\frac{i(i-1)+2}{2}}} \right) \\
= & \frac{1}{2^{n_{\mathsf{H}}}} \left(\rho^*\right), \qquad (5.4.3)
\end{aligned}$$

and thus refresh' is a strong $\rho' = 2^{-n_{\mathsf{H}}}(1 + (\rho^* - 1))$-universal hash function.

Corollary 5.4.15.
The algorithm refresh' as described above is a $(k, \epsilon_{\mathsf{ext}}^{\mathsf{refresh}'})$-extractor for $\epsilon_{\mathsf{ext}}^{\mathsf{refresh}'}$ given in Lemma 5.5.14 under the assumption that E *is an ideal cipher.*

Proof. By the leftover hash lemma we have that since refresh' is a $2^{-n_{\mathsf{H}}}(1+(\rho^*-1))$-universal hash function, it is also a $(k, \epsilon_{\mathsf{ext}}^{\mathsf{refresh}'})$-extractor for $\epsilon_{\mathsf{ext}}^{\mathsf{refresh}'} \leq \sqrt{2^{n_{\mathsf{H}}-k} + \rho^* - 1}$. Inputting $k \geq \gamma^*$ yields our result.

Putting this together with Lemma 5.4.8 yields

$$\epsilon_{\mathsf{r}} \leq \epsilon_{\mathsf{ext}}^{\mathsf{refresh}'} + \epsilon_{\mathsf{PRG}}.$$

5.5 Security of the ctr_drbg

We now turn to the security of the ctr_drbg. We will assume the block cipher E is a secure PRP unless otherwise specified.

Whenever the ctr_drbg updates the state it uses the bcup algorithm. This is not limited to next function; each of the ctr_drbg algorithms calls bcup. The bcup algorithm utilises the previous state and the block cipher in counter mode to generate the new state. Because of this, the split masking function of ctr_drbg is much simpler than its counterpart of the hash_drbg. In Definition 5.5.1 we define the split masking function M of the ctr_drbg.

5.5.1 Masking Function of ctr_drbg

Definition 5.5.1. For Statespace defined as in Definition 5.3.1, the ctr_drbg masking function is as follows:

$$\mathsf{M} := (\mathsf{M_I}, \mathsf{M_P}, \mathsf{M_R}) : \{0,1\}^{n_\mathsf{E}} \times \{0,1\}^{n_\mathsf{k}} \times \{0,1\}^{49} \longrightarrow \{0,1\}^{n_\mathsf{E}} \times \{0,1\}^{n_\mathsf{k}} \times \{0,1\}^{49},$$
$$\mathsf{M_I}(s) = (v', k', 1) \quad \text{where } v' \xleftarrow{\$} \{0,1\}^{n_\mathsf{E}}, k' \xleftarrow{\$} \{0,1\}^{n_\mathsf{k}},$$
$$\mathsf{M_P}(s) = (v', k', \mathsf{rc}) \quad \text{where } v' \xleftarrow{\$} \{0,1\}^{n_\mathsf{E}}, k' \xleftarrow{\$} \{0,1\}^{n_\mathsf{k}},$$
$$\mathsf{M_R}(s) = (v', k', \mathsf{rc}) \quad \text{where } v' \xleftarrow{\$} \{0,1\}^{n_\mathsf{E}}, k' \xleftarrow{\$} \{0,1\}^{n_\mathsf{k}}.$$

Lemma 5.5.2.
The ctr_drbg split masking function as defined in Definition 5.5.1 is idempotent.

Proof. The ctr_drbg split masking function acts the same in all situations and always overwrites the v-state and k-state, leaving the reseed counter untouched. Each successive application of the split masking function overwrites the states with samples from the same distribution, therefore the mask applied iteratively results in the same distribution. □

Lemma 5.5.3.
The split masking function of the ctr_drbg defined in Definition 5.5.1 is a $(\mathcal{D}, t, \epsilon_\mathsf{h})$-honest initialisation split masking function under the assumption that $\mathcal{D}$ is a legitimate distribution sampler outputting I_0 with minimum entropy $3 \cdot 2k = 6n_\mathsf{E}$-bits, that

E *is an ideal cipher,* seedgen *outputs a uniformly random seed* seed *from* Seedspace*, and where*

$$\epsilon_{\mathsf{h}} \leq 3^2 \cdot 2^{-n_{\mathsf{E}}/2} + \frac{3}{2^{n_{\mathsf{E}}}} + q^2 2^{n_{\mathsf{E}}-1} + \left(\frac{(qn_{\mathsf{E}})^2}{n_{\mathsf{E}}^2} - \frac{qn_{\mathsf{E}}}{n_{\mathsf{E}}}\right) 2^{-n_{\mathsf{E}}}.$$

Proof. Running Initialise, the setup algorithm uses the bcdf algorithm to extract entropy from the entropy input I_0. This value is then used by the bcup algorithm, which XORs it with initial values of k, v derived from $\mathsf{E}_{0^{n_\mathsf{k}}}(0^{n_{\mathsf{E}}})$. The proof proceeds by showing that bcdf acts as an extractor and outputs an n-bit string y indistinguishable from random. Once this is established, since the initial values (v_0, k_0) are a constant c, XORed with this value y and split into the required lengths, we arrive at the desired uniformly random initial working state.

Lemma 5.5.4.
The bcdf *algorithm is a* $(6n_{\mathsf{E}}, \epsilon_{\mathsf{h}})$*-extractor under the assumption that* E *is a secure PRP and the seed of the extractor is* nonce‖perstring *which are uniformly random strings.*

Proof. We proceed with a hybrid argument utilising the following games:

- **Game 0** is the unmodified bcdf algorithm,
- **Game 1** modifies bcdf so that instead of returning the concatenation of the constant keyed output of the bcc algorithm of length $n_{\mathsf{k}} + n_{\mathsf{E}}$, a new string is chosen uniformly at random from $\mathsf{U}_{n_{\mathsf{k}}+n_{\mathsf{E}}}$.
- **Game 2** the final output of the bcdf algorithm y is replaced with $y \overset{\$}{\leftarrow} \mathsf{U}_{n_{\mathsf{k}}+n_{\mathsf{E}}}$.

The bcdf algorithm first uses CBC-MAC with key k' to output (k, v), which is then used in CBC mode to output the requested bits, so we proceed with a hybrid argument, first switching out the intermediate (k, v) for uniformly random strings, followed by the final output produced by bcdf which is split and used as the initial state.

The bcdf algorithm runs CBC-MAC up to three times to generate enough output to be used as the intermediate k-state and v. A different initialisation vector is used, namely $\mathsf{E}_{k'}(1), \mathsf{E}_{k'}(2), \mathsf{E}_{k'}(3)$ on the same input. Since k' is fixed, these IVs are entirely predictable to the adversary but work to separate each instance by keeping them prefix-free. The downside to this is that a uniformly random choice of the intermediate value can result in a collision (albeit with low probability) while the construction will never collide since E is a permutation starting from different values.

Looking at the first output, under the assumption that E is an ideal block cipher and the input to bcdf is a source with minimum entropy $2k$ and with $L = 3, k = n_{\mathsf{E}}$, then, by [28, Theorem 1] the statistical distance between $\mathsf{bcc}(k', (1\|x))$ and the uniform distribution on $\{0,1\}^{n_{\mathsf{E}}}$ is

$$\sqrt{2^{n_{\mathsf{E}}-\mathbf{H}_\infty(x)} + O(2^{n_{\mathsf{E}}} \cdot (O(3^2/2^{2n_{\mathsf{E}}}))},$$

and since $L = 3 \leq 2^{n_{\mathsf{E}}/4}$ we have

$$\sqrt{2^{n_{\mathsf{E}}-\mathbf{H}_\infty(x)} + O(2^{n_{\mathsf{E}}} \cdot (O(3^2/2^{2n_{\mathsf{E}}}))} \leq 3 \cdot 2^{-n_{\mathsf{E}}/2}.$$

Since this is repeated three times, with different IVs the statistical distance between (v, k) and U_n is

$$|\Pr[\mathbf{G_0}(\mathcal{A}) = 1] - \Pr[\mathbf{G_1}(\mathcal{A}) = 1]| \leq 3^2 \cdot 2^{-n_{\mathsf{E}}/2} + \frac{3}{2^{n_{\mathsf{E}}}}. \tag{5.5.1}$$

The second part of the bcdf algorithm then uses this intermediate pair (k, v) with the same E in CBC mode with $L = 3$ and $IV = 0$. By [11, Theorem 17], we have

$$\begin{aligned}|\Pr[\mathbf{G_1}(\mathcal{A}) = 1] - \Pr[\mathbf{G_2}(\mathcal{A}) = 1]| \leq & 2\mathsf{Adv}_{\mathsf{E}}^{\mathsf{PRP}}(t, q) \\ & + q^2 2^{n_{\mathsf{E}}-1} + \left(\frac{(qn_{\mathsf{E}})^2}{n_{\mathsf{E}}^2} - \frac{qn_{\mathsf{E}}}{n_{\mathsf{E}}}\right) 2^{-n_{\mathsf{E}}},\end{aligned}$$

and since E is a random permutation, $\mathsf{Adv}_{\mathsf{E}}^{\mathsf{PRP}} = 0$, thus

$$\begin{aligned}|\Pr[\mathbf{G_1}(\mathcal{A}) = 1] - \Pr[\mathbf{G_2}(\mathcal{A}) = 1]| \leq & q^2 2^{n_{\mathsf{E}}-1} \\ & + \left(\frac{(qn_{\mathsf{E}})^2}{n_{\mathsf{E}}^2} - \frac{qn_{\mathsf{E}}}{n_{\mathsf{E}}}\right) 2^{-n_{\mathsf{E}}}.\end{aligned} \tag{5.5.2}$$

Putting Equations (5.5.1) and (5.5.2) together yields

$$\epsilon_{\mathsf{h}} = 3^2 \cdot 2^{-n_{\mathsf{E}}/2} + \frac{3}{2^{n_{\mathsf{E}}}} + q^2 2^{n_{\mathsf{E}}-1} + \left(\frac{(qn_{\mathsf{E}})^2}{n_{\mathsf{E}}^2} - \frac{qn_{\mathsf{E}}}{n_{\mathsf{E}}}\right) 2^{-n_{\mathsf{E}}}.$$

□

The proof of Lemma 5.5.3 follows directly from Lemma 5.5.4, since the split masking function of the ctr_drbg in the initialise function overwrites the v-state and k-state with uniformly random strings of length n_{E} and n_{k} respectively. Since bcdf outputs a string of length $n_{\mathsf{k}} + n_{\mathsf{E}}$ indistinguishable from a uniformly random string, which is then XORed with constants, resulting in the output state being distributed identically to the masked state. □

5.5.2 PRG Security of the next Function of the ctr_drbg

We now turn our attention to the output of the ctr_drbg next algorithm. We utilise the following definition of ctr_drbg.next PRG security:

Definition 5.5.5. Let $\mathsf{D_i}$ be an algorithm representing the next function, $\mathsf{M_I}(s'_0)$ be a masked input state, seed seed, $\ell_{i+1} \leq \ell_{\mathsf{max}}$ be the requested number of bits, and $\mathcal{A}_{\mathsf{PRG}}$ be any adversary running in time no more than t playing the game described in Figure 5.15. The game described in Figure 5.15 is derived from the resilence notion (from Definition 4.4.2) restricted to one call to next. We say that $\mathcal{A}_{\mathsf{PRG}}$ has advantage: has advantage

$$\mathsf{Adv}^{\mathsf{dist}}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D_i}(s_0, \ell_{i+1}), \mathsf{D_{i+1}}(s_0, \ell_{i+1})) := \left| \Pr\left[\mathcal{A}^{\mathsf{D_i}}_{\mathsf{PRG}} = 1\right] - \Pr\left[\mathcal{A}^{\mathsf{D_{i+1}}}_{\mathsf{PRG}} = 1\right]\right|.$$

```
$\underline{\mathsf{res}_1(\mathcal{A}_{\mathsf{PRG}}, \mathsf{D_i}, \mathsf{D_{i+1}}, \ell_{i+1})}$
$\mathsf{seed} \leftarrow \mathsf{seedgen}; \mathsf{b} \xleftarrow{\$} \{0,1\}$
$s'_0 \xleftarrow{\$} \{0,1\}^{n_{\mathsf{E}}+n_{\mathsf{k}}+49}$
$s_0 \leftarrow \mathsf{M}(s'_0)$
$(s_{i+1}, r_{i+1}) \leftarrow \mathsf{D_{i+b}}(\mathsf{seed}, \mathsf{M}(s'_0), \ell_{i+1})$
$\mathsf{b}^* \xleftarrow{\$} \mathcal{A}_{\mathsf{PRG}}((s_{i+1}, r_{i+1}), \mathsf{rc})$
return $\mathsf{b} == \mathsf{b}^*$
```

Figure 5.17: The security game for distinguishing between ctr_drbg.next and a random generator output and masked state.

Lemma 5.5.6.
Let s_i be a random variable on the Statespace*, then for any q adversary, $\mathcal{A}_{\mathsf{PRG}}$, and $\ell_{i+1} \leq \ell_{max}$,*

$$\mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{next}(s_i, \ell_{i+1}), (\mathsf{M}(s_{i+1}), \mathsf{U}_{\ell_{i+1}})) \leq \epsilon_{\mathsf{PRG}}$$

with

$$\epsilon_{\mathsf{PRG}} \leq 2\left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E}}\left(t, \left\lceil \frac{\ell_{i+1}}{n_{\mathsf{E}}} \right\rceil\right) + \left\lceil \frac{\ell_{i+1}}{n_{\mathsf{E}}} \right\rceil^2 2^{-n_{\mathsf{E}}}\right) + 2\left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E}}(t, 3) + 3^2 2^{-n_{\mathsf{E}}}\right).$$

Proof. The proof will proceed in two parts. First we will show that an adversary cannot distinguish between the real or masked output state, while always having real output. Following this, given masked output state the adversary cannot distinguish between real and random output.

We define three algorithms in Figure 5.18 that we will use for the hybrid game jumps. We omit parsing the seed for simplicity.

```
Alg. $\mathsf{D}_0(\mathsf{seed}, s_i, \ell_{i+1})$
parse $s_i$ as $(v_i, k_i, \mathsf{rc})$
$\mathsf{add} \leftarrow \mathsf{bcdf}(\mathsf{add}, n)$
$(k, v) \leftarrow \mathsf{bcup}(\mathsf{add}, k_i, v_i)$
$y \leftarrow \mathsf{null}$
while $\mathsf{Len}(y) < \ell_{i+1}$ do
    $v \leftarrow v + 1 \bmod 2^{n_{\mathsf{E}}}$
    $y \leftarrow y \| \mathsf{E}_k(v)$
$r_{i+1} \leftarrow \mathsf{L}_{\ell_{i+1}}(y)$
$(k_{i+1}, v_{i+1}) \leftarrow \mathsf{bcup}(\mathsf{add}, k, v)$
$\mathsf{rc} \leftarrow \mathsf{rc} + 1$
return $((v_{i+1}, k_{i+1}, \mathsf{rc}), r_{i+1})$
```

```
Alg. $\mathsf{D}_1(\mathsf{seed}, s_i, \ell_{i+1})$
parse $s_i$ as $(v_i, k_i, \mathsf{rc})$
$\mathsf{add} \leftarrow \mathsf{bcdf}(\mathsf{add}, n)$
$(k, v) \leftarrow \mathsf{bcup}(\mathsf{add}, k_i, v_i)$
$y \leftarrow \mathsf{null}$
while $\mathsf{Len}(y) < \ell_{i+1}$ do
    $v \leftarrow v + 1 \bmod 2^{n_{\mathsf{E}}}$
    $y \leftarrow y \| \mathsf{E}_k(v)$
$r_{i+1} \leftarrow \mathsf{L}_{\ell_{i+1}}(y)$
$(k_{i+1}, v_{i+1}) \leftarrow \mathsf{bcup}(\mathsf{add}, k, v)$
$\mathsf{rc} \leftarrow \mathsf{rc} + 1$
$s_{i+1} \leftarrow \mathsf{M}(s_{i+1})$
return $((v_{i+1}, k_{i+1}, \mathsf{rc}), r_{i+1})$
```

```
Alg. $\mathsf{D}_2(\mathsf{seed}, s_i, \ell_{i+1})$
parse $s_i$ as $(v_i, k_i, \mathsf{rc})$
$\mathsf{add} \leftarrow \mathsf{bcdf}(\mathsf{add}, n)$
$(k, v) \leftarrow \mathsf{bcup}(\mathsf{add}, k_i, v_i)$
$y \leftarrow \mathsf{null}$
while $\mathsf{Len}(y) < \ell_{i+1}$ do
    $v \leftarrow v + 1 \bmod 2^{n_{\mathsf{E}}}$
    $y \leftarrow y \| \mathsf{E}_k(v)$
$r_{i+1} \leftarrow \mathsf{L}_{\ell_{i+1}}(y)$
$r_{i+1} \xleftarrow{\$} \{0,1\}^{\ell_{i+1}}$
$(k_{i+1}, v_{i+1}) \leftarrow \mathsf{bcup}(\mathsf{add}, k, v)$
$\mathsf{rc} \leftarrow \mathsf{rc} + 1$
$s_{i+1} \leftarrow \mathsf{M}(s_{i+1})$
return $((v_{i+1}, k_{i+1}, \mathsf{rc}), r_{i+1})$
```

Figure 5.18: The algorithms $\mathsf{D}_0, \mathsf{D}_1$ and D_2 used in proving the PRG security of the ctr_drbg.next function.

We then have,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{next}(s_i,\ell_{i+1}),(\mathsf{M}(s_{i+1}),\mathsf{U}_{\ell_{i+1}})) &= \mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_0(s_i,\ell_{i+1}),\mathsf{D}_2(s_i,\ell_{i+1}))\\ &\leq \mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_0(s_i,\ell_{i+1}),\mathsf{D}_1(s_i,\ell_{i+1}))\\ &+ \mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_1(s_i,\ell_{i+1}),\mathsf{D}_2(s_i,\ell_{i+1})).\end{aligned}$$

The ctr_drbg.next algorithm updates the state using the bcup algorithm, which doesn't affect the distribution of the intermediary state when acting on a masked state. This is proved formally in Corollary 5.5.10. This intermediate state is processed in counter mode to produce the generator output which we capture in Lemma 5.5.7. Finally, the updated intermediate state is used in the bcup algorithm, which again uses counter mode to produce the updated states with three blocks of output, captured in Corollary 5.5.10.

Lemma 5.5.7.
Let s_i be as described in Definition 5.5.5 and let E be a secure block cipher. Then for any adversary working in time t, $\mathcal{A}_{\mathsf{PRG}}$, and $\ell_{i+1} \leq \ell_{max}$,

$$\mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_1(s_i,\ell_{i+1}),\mathsf{D}_2(s_i,\ell_{i+1})) \leq 2\left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E}}\left(t,\left\lceil\frac{\ell_{i+1}}{n_{\mathsf{E}}}\right\rceil\right) + \left\lceil\frac{\ell_{i+1}}{n_{\mathsf{E}}}\right\rceil^2 2^{-n_{\mathsf{E}}}\right).$$

Proof. By the prp/prf switching lemma (given in Lemma 5.5.8) and security of a block cipher in counter mode using a PRF (given in Theorem 5.5.9) [11, Proposition 8 and Theorem 13] we have

$$\begin{aligned}\mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_1(s_i,\ell_{i+1}),\mathsf{D}_2(s_i,\ell_{i+1})) &\leq 2\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{E}}\left(t,\left\lceil\frac{\ell_{i+1}}{n_{\mathsf{E}}}\right\rceil\right)\\ &\leq 2\left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E}}\left(t,\left\lceil\frac{\ell_{i+1}}{n_{\mathsf{E}}}\right\rceil\right) + \left\lceil\frac{\ell_{i+1}}{n_{\mathsf{E}}}\right\rceil^2 2^{-n_{\mathsf{E}}}\right).\end{aligned}$$

Lemma 5.5.8 (Proposition 8 of [11]).
For any permutation family P with length l,

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{P}}(t,q) \leq \mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{P}}(t,q) + q^2 2^{-l-1}.$$

Theorem 5.5.9 (Theorem 13 of [11]).
Suppose F is a PRF family with input length l and output length L. Then for any $t, q, \mu = \min(q'L, L2^l)$,

$$\mathsf{Adv}^{\mathsf{ror-cpa}}_{\mathsf{CTR[F]}}(t,q,\mu) \leq 2\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}}(t,q').$$

□

The jump between D_0 and D_1 is a simpler case of Lemma 5.5.7 where the number of blocks is 3, relating to $n = 3n_{\mathsf{E}}$. Formally:

Corollary 5.5.10.

Let s_i be as described in Definition 5.5.5, let E be a secure block cipher, then for any adversary working in time t, $\mathcal{A}_{\mathsf{PRG}}$, and $\ell_{i+1} \leq \ell_{max}$,

$$\mathsf{Adv}^{\mathsf{res}_1}_{\mathcal{A}_{\mathsf{PRG}}}(\mathsf{D}_0(s_i, \ell_{i+1}), \mathsf{D}_1(s_i, \ell_{i+1})) \leq 2\left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E}}(t, 3) + 3^2 2^{-n_{\mathsf{E}}}\right).$$

Proof. This follows directly from Lemma 5.5.7. □

Putting together Lemma 5.5.7 and corollary 5.5.10 concludes the proof. □

5.5.3 Variable-Output Robustness of the ctr_drbg

Theorem 5.5.11.

Let E be an ideal cipher, let the ctr_drbg be as described above, with legitimate distribution sampler $\mathcal{D}$. Then the hash_drbg is $(t', \ell_{max}, \ell_{total}, \mathsf{q}_{\mathcal{D}}, \mathsf{q}_{\mathsf{S}}, \gamma^*, \epsilon_{\mathsf{rob}})$-variable outupt robust witnessed by the $(\mathcal{D}, t, \epsilon_{\mathsf{h}})$-honest idempotent split masking function M where

$$\epsilon_{\mathsf{rob}} \leq \epsilon_{\mathsf{h}} + \max_{\substack{(\ell_1,\ldots,\ell_{\mathsf{q}}):\\ \sum_{i=1}^{\mathsf{q}} \ell_i \leq \ell_{total}}} \left(\sum_{i=1}^{q} (\epsilon_{\mathsf{p}} + \epsilon_{\mathsf{r}})\right),$$

for ϵ_{p} and ϵ_{r} given in the following lemmas.

Proof. The proof proceeds as normal by proving the ctr_drbg satisfies both variable-output preserving and variable output-recovering security.

Lemma 5.5.12.

Let E *be an ideal cipher, let the* $\mathsf{ctr\_drbg}$ *be as described above. Then the* $\mathsf{ctr\_drbg}$ *has* $(t, \ell_{max}, \epsilon_{\mathsf{p}})$*-variable output preserving security, witnessed by the masking function* M *defined in Definition 5.5.1 and for:*

$$\epsilon_{\mathsf{p}} \leq 2^{1-n_{\mathsf{E}}} \left(\left\lceil \frac{\ell_{i+1}}{n_{\mathsf{E}}} \right\rceil^2 + 3^2(d+1) \right).$$

Proof. The proof proceeds by using a hybrid argument. First we show that we can replace the intermediate state s_d after the adversarial refresh queries with a masked state, followed by using the earlier result of Lemma 5.5.6 to show that the final challenge is indistinguishable from random output and a masked state.

Since the adversary has knowledge of the additional input add, the output s of $\mathsf{bcdf}(\mathrm{I}\|\mathsf{add}, n)$ is predictable for each adversarial refresh I. The $\mathsf{ctr\_drbg.refresh}$ algorithm then utilises this value in the bcup algorithm, which XORs the input s with the updated state. By Corollary 5.5.10 we know that for random state s_i and predictable input x, the bcup algorithm returns a state indistinguishable from random. Since the adversary is allowed to make d adversarial refresh calls this becomes

$$d \left(\mathsf{Adv}_{\mathsf{E}}^{\mathsf{PRP}}(t,3) + 3^2 2^{-n_{\mathsf{E}}} \right) = 3^2 d 2^{1-n_{\mathsf{E}}}.$$

Putting this together with Lemma 5.5.6 yields

$$\epsilon_{\mathsf{p}} = 3^2 d 2^{1-n_{\mathsf{E}}} + 2^{1-n_{\mathsf{E}}} \left(\left\lceil \frac{\ell_{i+1}}{n_{\mathsf{E}}} \right\rceil^2 + 3^2 \right) = 2^{1-n_{\mathsf{E}}} \left(\left\lceil \frac{\ell_{i+1}}{n_{\mathsf{E}}} \right\rceil^2 + 3^2(d+1) \right).$$

□

Lemma 5.5.13.

Let E *be an ideal cipher, let the* $\mathsf{ctr\_drbg}$ *be as described in Definition 5.3.1, with legitimate distribution sampler* $\mathcal{D}$*. Then the* $\mathsf{hash\_drbg}$ *has* $(t, \mathsf{q}_{\mathcal{D}}, \ell_{max}, \gamma^*, \epsilon_{\mathsf{r}})$*-variable output recovering security, witnessed by the masking function* M *defined in Definition 5.5.1 and for:*

$$\epsilon_{\mathsf{r}} := \epsilon_{\mathsf{ext}}^{\mathsf{refresh}'} + \epsilon_{\mathsf{PRG}}.$$

Proof. The proof proceeds by using a hybrid argument. First we show that we can replace the intermediate state after the refresh queries with a masked state, followed by using the earlier result of Lemma 5.5.6 to show that the final challenge is indistinguishable from random output and a masked state.

If we modify the refresh algorithm to take as input the entire string of refresh values as an on-line extractor, call it $\mathsf{refresh}'(\mathsf{seed}, s_i, \mathrm{I}')$ for $\mathrm{I}' := \mathrm{I}_1, \ldots, \mathrm{I}_d$, we can prove that this new algorithm is a universal hash function and employ the leftover hash lemma. Recall that m in the context of bcc is $m := \mathsf{Len}(x)/n_{\mathsf{E}} \in \mathbb{N}_{>1}$, $m > 1$ since the first block is always the IV, while the second contains two fields dictating the size of the input/output, followed by the input which may span several blocks.

Lemma 5.5.14.
The algorithm refresh' *as described above is a* ρ*-universal hash function, assuming that* E *is an ideal cipher and for:*

$$\rho := \sum_{i=2}^{d} \left(\Pr\left[\mathsf{coll}|\mathbf{Ev_i}\right]\right) + \epsilon^{\mathsf{bcdf}}_{\mathsf{coll}} = \frac{1}{2^{3n_{\mathsf{E}}}} \left(\rho'\right).$$

Proof. Let $\mathrm{I}'_0 \neq \mathrm{I}'_1$, with $\mathrm{I}'_j := (\mathrm{I}_{j;1} \ldots, \mathrm{I}_{j;d})$. We split into several cases. Let $\mathbf{Ev_i}$ be the event corresponding to where I'_0 and I'_1 only differ in the first i blocks, with the edge cases being $\mathbf{Ev_1}$ and $\mathbf{Ev_d}$. The former, $\mathbf{Ev_1}$ is where $\mathsf{refresh0}'$ and $\mathsf{refresh1}'$ differ in the first block alone, while the latter, $\mathbf{Ev_d}$ refers to differences in all d blocks.

In $\mathbf{Ev_1}$, the probability of a collision is precisely the probability that either there is a collision in the output of bcdf, with the probability taken over the choice of add. To begin, we focus on the bcc algorithm, noting that if $m = 2$ then a collision is impossible since k is constant and by assumption $\mathrm{I}_{0;1} \neq \mathrm{I}_{1;1}$. We will assume $m > 2$. Building upon this, the probability of a collision is therefore

$$\left(\frac{m-2}{2^{n_{\mathsf{E}}}}\right) \left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E}_{\mathsf{k}'}}(t, m) + \frac{m^2}{2^{n_{\mathsf{E}}-1}}\right),$$

making use of the PRP/PRF switching lemma. Since this is repeated at most three times with a different IV/first block, this becomes

$$\epsilon^{\mathsf{bcc}}_{\mathsf{coll0}} := \left(\frac{m-2}{2^{n_{\mathsf{E}}}}\right)^3 \left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E}_{\mathsf{k}'}}(t, m) + \frac{m^2}{2^{n_{\mathsf{E}}-1}}\right)^3. \tag{5.5.3}$$

If this collision occurs, then the remaining parts of bcdf will also collide, resulting in a collision. However, if this does not collide, there is still the possibility that the final three blocks output from CBC mode will collide, which occurs with probability:

$$\epsilon^{\mathsf{bcc}}_{\mathsf{coll1}} := \left(\frac{1}{2^{3n_{\mathsf{E}}}}\right)\left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E_k}}(t,3) + \frac{3^2}{2^{n_{\mathsf{E}}-1}}\right)^3, \tag{5.5.4}$$

and putting together with Equation (5.5.3) yields the collision probability of bcdf with distinct inputs to be

$$\begin{aligned}\epsilon^{\mathsf{bcdf}}_{\mathsf{coll}} := \epsilon^{\mathsf{bcc}}_{\mathsf{coll0}} + \epsilon^{\mathsf{bcc}}_{\mathsf{coll1}} &= \left(\frac{m-2}{2^{n_{\mathsf{E}}}}\right)^3\left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E_{k'}}}(t,m) + \frac{m^2}{2^{n_{\mathsf{E}}-1}}\right)^3 \\ &+ \left(\frac{1}{2^{3n_{\mathsf{E}}}}\right)\left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E_k}}(t,3) + \frac{3^2}{2^{n_{\mathsf{E}}-1}}\right)^3 \\ &= \frac{1}{2^{3n_{\mathsf{E}}}}\left((m-2)^3\right)\left(\mathsf{Adv}^{\mathsf{PRP}}_{\mathsf{E_{k'}}}(t,m) + \frac{m^2}{2^{n_{\mathsf{E}}-1}}\right)^3.\end{aligned} \tag{5.5.5}$$

We note that in general m will be reasonably small, but note that Table 5.5 states the maximum input size of a single block of input is 2^{35}. In the case of $\mathbf{Ev_1}$, if there is not a collision in the output of bcdf, there cannot be an output collision since the current value of the state is identical, meaning the output of bcup will be distinct, since E_{k_0} is a permutation called on the same input and key.

Now that we have dealt with the case $\mathbf{Ev_1}$ we can look to the general case of $\mathbf{Ev_i}$. The general case can be split as follows:

1. Similarly to $\mathbf{Ev_1}$, the output of bcdf may collide in the first refresh call, followed by a collision in the output of bcdf for all remaining $i-1$ distinct inputs.
2. On input block k, which has distinct k_k, v_k and input block $\mathrm{I}_{j;k}$ there is a collision in the output of bcup, followed by a collision in the output of bcdf for all remaining $i-k$ distinct inputs.

The first case is relatively simple, since it requires a collision in bcdf for i distinct input pairs, which is $(\epsilon^{\mathsf{bcdf}}_{\mathsf{coll}})^i$ for $\epsilon^{\mathsf{bcdf}}_{\mathsf{coll}}$ given in Equation (5.5.5).

The second case requires us to calculate the probability that the output of bcup collides with distinct input values. Since we have assumed that E_k is an ideal cipher, this is precisely the probability that (after XORing two distinct values s) there are three collisions which will be exactly $1/2^{n_{\mathsf{E}}}$. For a collision in the final output of refresh' we now require a collision in the remaining $i-k$ outputs from bcdf, which together yields:

$$\begin{aligned}\Pr\left[\mathsf{coll}|\mathbf{Ev_i}\right] &= \sum_{k=2}^{i-1}\left(\frac{\left(\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}}\right)^{i-k}}{2^{3n_{\mathsf{E}}}}\right) + (\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}})^{i} \\ &= \frac{1}{2^{3n_{\mathsf{E}}}}\left(\sum_{k=1}^{i-2}(\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}})^{k}\right) + (\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}})^{i}.\end{aligned} \tag{5.5.6}$$

By taking the geometric progression sum this becomes:

$$\begin{aligned}&= \frac{1}{2^{3n_{\mathsf{E}}}}\left(\frac{(\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}})^{i-1} - \epsilon_{\mathsf{coll}}^{\mathsf{bcdf}}}{\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}} - 1}\right) + (\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}})^{i} \\ &= \frac{1}{2^{3n_{\mathsf{E}}}}\left(\frac{2^{3n_{\mathsf{E}}}(\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}} - 1)(\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}})^{i} + (\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}})^{i-1} - \epsilon_{\mathsf{coll}}^{\mathsf{bcdf}}}{\epsilon_{\mathsf{coll}}^{\mathsf{bcdf}} - 1}\right),\end{aligned} \tag{5.5.7}$$

for $k \geq 2$. Therefore, the total collision probability will be

$$\rho := \sum_{i=2}^{d}\left(\Pr\left[\mathsf{coll}|\mathbf{Ev_i}\right]\right) + \epsilon_{\mathsf{coll}}^{\mathsf{bcdf}} = \frac{1}{2^{3n_{\mathsf{E}}}}\left(\rho'\right),$$

and thus refresh' is a $\rho = 2^{-n}(1 + (\rho' - 1))$-universal hash function. ☐

Corollary 5.5.15.
The algorithm refresh' as described above is a $(k, \epsilon_{\mathsf{ext}}^{\mathsf{refresh}'})$-extractor for $\epsilon_{\mathsf{ext}}^{\mathsf{refresh}'}$ given in Lemma 5.5.14 under the assumption that E *is an ideal cipher.*

Proof. By the leftover hash lemma we have that since refresh' is a $2^{-n}(1 + (\rho' - 1))$-universal hash function, it is also a $(k, \epsilon_{\mathsf{ext}}^{\mathsf{refresh}'})$-extractor for $\epsilon_{\mathsf{ext}}^{\mathsf{refresh}'} \leq \sqrt{2^{n-k} + \rho' - 1}$. Inputting $k \geq \gamma^*$ yields our result. ☐

Putting this together with Lemma 5.5.6 yields

$$\epsilon_{\mathsf{r}} \leq \epsilon_{\mathsf{ext}}^{\mathsf{refresh}'} + \epsilon_{\mathsf{PRG}}.$$

☐

☐

5.6 Conclusions

We have presented the two NIST VOPWIs in the format of our updated security model from Chapter 4 and have analysed their security under some ideal assumptions and concluded that they meet the notion of variable-output robustness. This essentially means that for a reasonable family of adversarial entropy sources, assuming the underlying primitives are ideal, the NIST generators do support a very reasonable level of security.

However, to highlight one of the largest chasms between theory and practice in this area, we require some very strong assumptions on the randomness of the seed of the generator that goes against the NIST framework and in general is not implemented in practice. In practice these values will contain little entropy, or possibly not be included, which is a problem in a theoretical setting where the adversarial entropy source is able to find distributions that are predictable to an adversary. In practice, however, the family of entropy sources that NIST recommends, and in fact requires to be used with the generators, is far more restricted and in effect may negate this problem entirely, especially since in the designs there are additional health checking mechanisms that are not included in the theoretical security modelling.

This is an unfortunate gap that is very difficult to address due in part to the difficulty in accurately measuring entropy, however the theoretical security that we have proved is still valid, if overly restrictive. Our result has allowed us to provide a meaningful result for the NIST generators, which, at the time of writing had received no formal analysis, possibly due to their "incompatibility" with the current models.

Chapter 6

Conclusion

Contents

Here we give an overview of the findings of the thesis, including discussion of several areas, and possible future directions in the field.

6.1 Overview

In this thesis we have considered and analysed several PWI constructions. In Chapter 3 of the thesis we commented on the current sponge-based PWIs in the literature, and identified where we thought improvements could be made, specifically the number of calls to the underlying permutation needed in the next algorithm. We then presented a more efficient forward security mechanism to use in the next algorithm, presented in the format of the security model of [29]. We then proved that this new sponge-like PWI, named Reverie, satisfied the relevant security notions despite departing slightly from the sponge construction. We did this by utilising the H-coefficient technique. Specifically, by formalising an adversary's queries into good and bad transcripts, we were able to arrive at a bound for security. By incorporating this updated next algorithm, we significantly reduced the number of calls to the underlying permutation needed from $t + 1$ calls to 1 call, making our PWI more suitable for constrained environments where minimising the calls to the underlying primitive is essential. We still had to rely on the assumption that the seed of the generator was generated uniformly at random to avoid non-negligible adversarial advantage from the adversarial entropy source.

We then focused primarily on the security model of Dodis et. al. [29] in Chapter 4, and extended the model in several ways, primarily to allow for an adversary to request varying amounts of output from the PWI and basing security on the *amount* of output as opposed to the number of queries. We defined PWIs that satisfied this behaviour as variable-output PWIs (VOPWIs) and updated the notion of robustness from [29] accordingly, including an update to the simpler notions of preserving and recovering security. We then proved an analogy of the composition theorem, proving that if a VOPWI satisfied the notions of variable-output preserving and variable-output recovering security, then the VOPWI satisfies the notion of variable-output robustness.

Although we identified a large gap between theory and practice regarding the uniformity of the seed of the generator, we were not able to find a way to overcome this problem. This is in part due to the difficulty in measuring entropy accurately and efficiently. We did make modifications to the generation of the seed and initial state to reflect on real world PWI specifications, making steps towards the idea that the initial state in practice draws entropy from the same entropy source that the PWI later relies on for fresh entropy.

We then applied our updated security model to two of the NIST PWIs, namely the hash_drbg and ctr_drbg in Chapter 5. The updated security model allows us to accurately model the generators and their subroutines that allow for differing amounts of output per next call. We prove that under some fairly strong assumptions, including uniform seeds, which is not the case in practice, both generators satisfy the notion of variable-output robustness. We prove this by proving that each generator satisfies the updated notions of variable-output preserving security and variable-output recovering security. Unfortunately the bounds are fairly complex, which often is the case when analysing real world generators.

6.2 Future Work

There were several other extensions we would have liked to have made to our security model. One instance would be to investigate restricting the number of next requests or output from a PWI since the last entropy input. This in effect would limit and perhaps simplify the possible requests an adversary acting in our variable-output security model could make, but would reflect specifications in real world generators, including the NIST generators, which do restrict the number of calls and output per recent entropy injection. We would like to investigate combining our extensions with the later security model extension of [30]. Since that model allows for a modular approach to constructing a robust PWI secure against premature get-next queries, it may be possible with small adjustments to extend to this setting.

We would also like to investigate the notion of a variable seed; the NIST generators allow for additional information from requesting applications to be supplied to the generator when output is requested. This would possibly be a step closer to less reliance on the common assumption of having a uniform seed generated during initialisation. This variable seed could perhaps be a separate entropy source, which could similarly be modelled as adversarial but, again, restricted in terms of communication with the other adversaries.

Another interesting extension would be to model two or more entropy sources in the same adversary model as [29], in the hopes that due to their inability to communicate and collude may negate the need for a seed.

We believe there are interesting options for extending the sponge-based design, possibly beyond the notion of a parazoa using, for example, the "sum" of two permutations or even the same permutation with domain separation, as suggested by Bellare, Krovetz and Rogaway in [12]:

$$\begin{aligned} \mathsf{XoP}^{\pi_1,\pi_2}(x) &:= \pi_1(x) \oplus \pi_2(x), \\ \mathsf{XoP}'^{\pi}(x) &:= \pi(0\|x) \oplus \pi(1\|x). \end{aligned}$$

Patarin showed that the single permutation with domain separation achieved similar security as XoP [48]. More recently, the encrypted Davies-Meyer (EDM) construction has been considered [24, 43]. Possibly this construction could be utilised.

It is possible that our proofs could be improved in several ways, especially given the complexity of the bounds. The proofs of Reverie are not made using the notion of masking functions, which would be a reasonable improvement. It may also be possible to utilise Patarin's mirror theory in conjunction with his H-coefficient technique, similar to [49]. We also argue that since the refresh algorithm in the constructions acts as a universal hash function over several entropy inputs, it would be nicer to prove or construct a refresh algorithm that is a multiple-sampler extractor, as defined in [5].

Bibliography

[1] Michel Abdalla, Sonia Belaïd, David Pointcheval, Sylvain Ruhault, and Damien Vergnaud. Robust pseudo-random number generators with input secure against side-channel attacks. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15: 13th International Conference on Applied Cryptography and Network Security*, volume 9092 of *Lecture Notes in Computer Science*, pages 635–654, New York, NY, USA, June 2–5, 2015. Springer, Heidelberg, Germany.

[2] Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. Security of keyed sponge constructions using a modular proof approach. In Gregor Leander, editor, *Fast Software Encryption – FSE 2015*, volume 9054 of *Lecture Notes in Computer Science*, pages 364–384, Istanbul, Turkey, March 8–11, 2015. Springer, Heidelberg, Germany.

[3] Elena Andreeva, Bart Mennink, and Bart Preneel. The parazoa family: generalizing the sponge hash functions. *International Journal of Information Security*, 11(3):149–165, Jun 2012.

[4] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels, editors, *ACM CCS 05: 12th Conference on Computer and Communications Security*, pages 203–212, Alexandria, Virginia, USA, November 7–11, 2005. ACM Press.

[5] Boaz Barak, Russell Impagliazzo, and Avi Wigderson. Extracting randomness using few independent sources. In *45th Annual Symposium on Foundations of*

Computer Science, pages 384–393, Rome, Italy, October 17–19, 2004. IEEE Computer Society Press.

[6] Boaz Barak, Ronen Shaltiel, and Eran Tromer. True random number generators secure in a changing environment. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 166–180, Cologne, Germany, September 8–10, 2003. Springer, Heidelberg, Germany.

[7] Elaine Barker and John. Recommendation for random number generation using deterministic random bit generators, SP800-90A. `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf`, June 2015.

[8] Elaine Barker and John Kelsey. Recommendation for random bit generator (RBG) constructions, SP800-90C. `http://csrc.nist.gov/publications/drafts/800-90/sp800_90c_second_draft.pdf`, April 2016.

[9] Elaine Barker and John Kelsey. Recommendation for the entropy sources used for random bit generation, SP800-90B. `http://csrc.nist.gov/publications/drafts/800-90/sp800-90b_second_draft.pdf`, Jan 2016.

[10] Andrew Becherer, Alex Stamos, and Nathan Wilcox. Cloud computer security: Raining on the trendy new parade, Blackhat presentation. `https://www.nccgroup.trust/globalassets/resources/us/presentations/cloud-blackhat-2009-isec.pdf`, July 2009.

[11] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *38th Annual Symposium on Foundations of Computer Science*, pages 394–403, Miami Beach, Florida, October 19–22, 1997. IEEE Computer Society Press.

[12] Mihir Bellare, Ted Krovetz, and Phillip Rogaway. Luby-Rackoff backwards: Increasing security by making block ciphers non-invertible. In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 266–280, Espoo, Finland, May 31 – June 4, 1998. Springer, Heidelberg, Germany.

[13] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93: 1st Con-*

ference on Computer and Communications Security, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press.

[14] Mihir Bellare and Phillip Rogaway. *Introduction to modern cryptography.* 2005.

[15] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

[16] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, 2007.

[17] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197, Istanbul, Turkey, April 13–17, 2008. Springer, Heidelberg, Germany.

[18] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge-based pseudo-random number generators. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 33–47, Santa Barbara, CA, USA, August 17–20, 2010. Springer, Heidelberg, Germany.

[19] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

[20] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM Journal on Computing*, 13(4):850–864, 1984.

[21] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography.* `https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_4.pdf`, 2018. Accessed: 13/03/2018.

[22] J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143 – 154, 1979.

[23] Shan Chen and John P. Steinberger. Tight security bounds for key-alternating ciphers. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 327–350, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.

[24] Benoît Cogliati and Yannick Seurin. EWCDM: An efficient, beyond-birthday secure, nonce-misuse resistant MAC. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 121–149, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany.

[25] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Heidelberg, Germany.

[26] D.W. Davies and W.L. Price. Digital signatures, an update. In *Proceedings of International Conference On Computer Communications*, pages 843–847, October 1984.

[27] Anand Desai, Alejandro Hevia, and Yiqun Lisa Yin. A practice-oriented treatment of pseudorandom number generators. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 368–383, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

[28] Yevgeniy Dodis, Rosario Gennaro, Johan Håstad, Hugo Krawczyk, and Tal Rabin. Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In Matthew Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 494–510, Santa Barbara, CA, USA, August 15–19, 2004. Springer, Heidelberg, Germany.

[29] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergnaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 647–658, Berlin, Germany, November 4–8, 2013. ACM Press.

[30] 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 Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 37–54, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[31] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[32] Christina Garman, Kenneth G. Paterson, and Thyla Merwe. Attacks only get better: Password recovery attacks against rc4 in tls, 2015.

[33] Peter Gazi and Stefano Tessaro. Provably robust sponge-based PRNGs and KDFs. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 87–116, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[34] Vipul Goyal, Adam O'Neill, and Vanishree Rao. Correlated-input secure hash functions. In Yuval Ishai, editor, *TCC 2011: 8th Theory of Cryptography Conference*, volume 6597 of *Lecture Notes in Computer Science*, pages 182–200, Providence, RI, USA, March 28–30, 2011. Springer, Heidelberg, Germany.

[35] Zvi Gutterman and Dahlia Malkhi. Hold your sessions: An attack on Java session-id generation. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 44–57, San Francisco, CA, USA, February 14–18, 2005. Springer, Heidelberg, Germany.

[36] Iftach Haitner, Thomas Holenstein, Omer Reingold, Salil P. Vadhan, and Hoeteck Wee. Universal one-way hash functions via inaccessible entropy. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume

6110 of *Lecture Notes in Computer Science*, pages 616–637, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany.

[37] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 205–220. USENIX Association, 2012.

[38] Shoichi Hirose. Security analysis of DRBG using HMAC in NIST SP 800–90. In Kyo-Il Chung, Kiwook Sohn, and Moti Yung, editors, *WISA 08: 9th International Workshop on Information Security Applications*, volume 5379 of *Lecture Notes in Computer Science*, pages 278–291, Jeju Island, Korea, September 23–25, 2009. Springer, Heidelberg, Germany.

[39] Daniel Hutchinson. A robust and sponge-like PRNG with improved efficiency. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*, volume 10532 of *Lecture Notes in Computer Science*, pages 381–398, St. John's, NL, Canada, August 10–12, 2016. Springer, Heidelberg, Germany.

[40] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition.* CRC Press, 2014.

[41] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, whit is right. Cryptology ePrint Archive, Report 2012/064, 2012. `http://eprint.iacr.org/2012/064`.

[42] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *TCC 2004: 1st Theory of Cryptography Conference*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39, Cambridge, MA, USA, February 19–21, 2004. Springer, Heidelberg, Germany.

[43] Bart Mennink and Samuel Neves. Encrypted davies-meyer and its dual: Towards optimal security using mirror theory. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 556–583, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[44] Bart Mennink, Reza Reyhanitabar, and Damian Vizár. Security of full-state keyed sponge and duplex: Applications to authenticated encryption. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 465–489, Auckland, New Zealand, November 30 – December 3, 2015. Springer, Heidelberg, Germany.

[45] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *21st Annual ACM Symposium on Theory of Computing*, pages 33–43, Seattle, WA, USA, May 15–17, 1989. ACM Press.

[46] Noam Nisan and David Zuckerman. Randomness is linear in space. *J. Comput. Syst. Sci.*, 52(1):43–52, February 1996.

[47] Jacques Patarin. The "coefficients H" technique. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography, 15th International Workshop, SAC 2008, Sackville, New Brunswick, Canada, August 14-15, Revised Selected Papers*, volume 5381 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 2008.

[48] Jacques Patarin. Introduction to mirror theory: Analysis of systems of linear equalities and linear non equalities for cryptography. Cryptology ePrint Archive, Report 2010/287, 2010. `http://eprint.iacr.org/2010/287`.

[49] Jacques Patarin. Security in $O(2^n)$ for the xor of two random permutations—proof with the standard H technique. Cryptology ePrint Archive, Report 2013/368, 2013. `http://eprint.iacr.org/2013/368`.

[50] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: A synthetic approach. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378, Santa Barbara, CA, USA, August 22–26, 1994. Springer, Heidelberg, Germany.

[51] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 487–506, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

[52] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *ISOC Network and Distributed System Security Symposium – NDSS 2010*, San Diego, CA, USA, February 28 – March 3, 2010. The Internet Society.

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

[54] Victor Shoup. *A Computational Introduction to Number Theory and Algebra.* Cambridge University Press, New York, NY, USA, 2 edition, 2009.

[55] Thomas Shrimpton and R. Seth Terashima. A provable-security analysis of Intel's secure key RNG. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 77–100, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

[56] Robert S. Winternitz. Producing a one-way hash function from DES. In David Chaum, editor, *Advances in Cryptology – CRYPTO'83*, pages 203–207, Santa Barbara, CA, USA, 1983. Plenum Press, New York, USA.

[57] A. C. Yao. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 80–91, Nov 1982.

[58] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17: 24th Conference on Computer and Communications Security*, pages 2007–2020, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[59] Yu Yu, François-Xavier Standaert, Olivier Pereira, and Moti Yung. Practical leakage-resilient pseudorandom generators. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10: 17th Conference on Computer and Communications Security*, pages 141–151, Chicago, Illinois, USA, October 4–8, 2010. ACM Press.