5 Pseudorandom Generators

One-time pad requires a key that's as long as the plaintext. Let's forget that we know about this limitation. Suppose Alice & Bob share only a short λ-bit secret k, but they want to encrypt a 2λ-bit plaintext m. They don't know that (perfect) one-time secrecy is impossible in this setting (Exercise 2.11), so they try to get it to work anyway using the following reasoning:

- The only encryption scheme they know about is one-time pad, so they decide that the ciphertext will have the form $c = m \oplus \boxed{??}$. This means that the unknown value $\boxed{??}$ must be 2λ bits long.
- In order for the security of one-time pad to apply, the unknown value $\boxed{??}$ should be uniformly distributed.
- The process of obtaining the unknown value $\boxed{??}$ from the shared key k should be *deterministic*, so that the sender and receiver compute the same value and decryption works correctly.

Let G denote the process that transforms the key k into this mystery value. Then $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$, and the encryption scheme is $\textsf{Enc}(k, m) = m \oplus G(k)$.

It is not hard to see that if G is a deterministic function, then there are only 2^λ possible outputs of G, so the distribution of $G(k)$ cannot be uniform in $\{0,1\}^{2\lambda}$. We therefore cannot argue that the scheme is secure in the same way as one-time pad.

However, what if the distribution of $G(k)$ values is not perfectly uniform but only "close enough" to uniform? Suppose no polynomial-time algorithm can distinguish the distribution of $G(k)$ values from the uniform distribution. Then surely this ought to be "close enough" to uniform for practical purposes. This is exactly the idea of **pseudorandomness.** It turns out that if G has a pseudorandomness property, then the encryption scheme described above is actually secure (against polynomial-time adversaries, in the sense discussed in the previous chapter).

5.1 Definitions

A **pseudorandom generator (PRG)** is a deterministic function G whose outputs are longer than its inputs. When the input to G is chosen uniformly at random, it induces a certain distribution over the possible output. As discussed above, this output distribution cannot be uniform. However, the distribution is *pseudorandom* if it is **indistinguishable from the uniform distribution.** More formally:

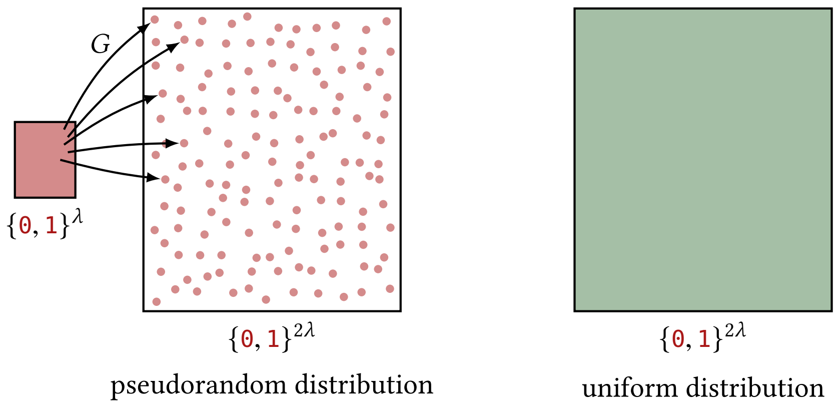
Definition 5.1 (PRG security) *Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be a deterministic function with $\ell > 0$. We say that G is a **secure pseudorandom generator (PRG)** if $\mathcal{L}^G_{\text{prg-real}} \approx \mathcal{L}^G_{\text{prg-rand}}$, where:*

```
$\mathcal{L}^G_{\text{prg-real}}$
QUERY():
  $s \leftarrow \{0,1\}^\lambda$
  return $G(s)$
```

```
$\mathcal{L}^G_{\text{prg-rand}}$
QUERY():
  $r \leftarrow \{0,1\}^{\lambda+\ell}$
  return $r$
```

*The value ℓ is called the **stretch** of the PRG. The input to the PRG is typically called a **seed**.*

Below is an illustration of the distributions sampled by these libraries, for a **length-doubling** ($\ell = \lambda$) PRG (not drawn to scale) :



$\mathcal{L}_{\text{prg-real}}$ samples from distribution of red dots, by first sampling a uniform element of $\{0,1\}^\lambda$ and performing the action of G on that value to get a red result in $\{0,1\}^{2\lambda}$. The other library $\mathcal{L}_{\text{prg-rand}}$ directly samples the uniform distribution on $\{0,1\}^{2\lambda}$ (in green above).

To understand PRGs, you must simultaneously appreciate two ways to compare the PRG's output distribution with the uniform distribution:

- From a *relative* perspective, the PRG's output distribution is tiny. Out of the $2^{2\lambda}$ strings in $\{0,1\}^{2\lambda}$, only 2^λ are possible outputs of G. These strings make up a $2^\lambda/2^{2\lambda} = 1/2^\lambda$ fraction of $\{0,1\}^{2\lambda}$ — a **negligible fraction!**

- From an *absolute* perspective, the PRG's output distribution is huge. There are 2^λ possible outputs of G, which is an **exponential amount!**

The illustration above only captures the *relative* perspective (comparing the red dots to the entire extent of $\{0,1\}^{2\lambda}$), so it can lead to some misunderstanding. Just looking at this picture, it is hard to imagine how the two distributions could be indistinguishable. How could a calling program *not* notice whether it's seeing the whole set or just a negligible fraction of the whole set? Well, if you run in polynomial-time in λ, then 2^λ and $2^{2\lambda}$ are both so enormous that it doesn't really matter that one is vastly bigger than the other. The relative *sizes* of the distribution don't really help distinguish, since it is not a viable strategy for the distinguisher to "measure" the size of the distribution it's sampling.

Consider: there are about 2^{75} molecules in a teaspoon of water, and about $2^{2 \cdot 75}$ molecules of water in Earth's oceans. Suppose you dump a teaspoon of water into the ocean and let things mix for a few thousand years. Even though the teaspoon accounts for only $1/2^{75}$ of the ocean's contents, that doesn't make it easy to keep track of all 2^{75} water molecules that originated in the teaspoon! If you are small enough to see individual water molecules, then a teaspoon of water looks as big as the ocean.

Discussion & Pitfalls

- Do not confuse the interface of a PRG (it takes in a seed as input) with the interface of the security libraries $\mathcal{L}_{\text{prg-}\star}$ (their QUERY subroutine doesn't take any input)! A PRG is indeed an algorithm into which you can feed any string you like. However, **security is only guaranteed** when the PRG is being used exactly as described in the security libraries — in particular, when the seed is chosen uniformly/secretly and not used for anything else.

  Nothing prevents a user from putting an adversarially-chosen s into a PRG, or revealing a PRG seed to an adversary, etc. You just get no security guarantee from doing it, since it's not the situation reflected in the PRG security libraries.

- It doesn't really make sense to say that "`0010110110` is a random string" or "`0000000001` is a pseudorandom string." Randomness and pseudorandomness are **properties of the *process* used to generate a string,** not properties of the individual strings themselves. When we have a value $z = G(s)$ where G is a PRG and s is chosen uniformly, you could say that z was "chosen pseudorandomly." You could say that the output of some process is a "pseudorandom distribution." But it is slightly sloppy (although common) to say that a string z "is pseudorandom".

- There are common statistical tests you can run, which check whether some data has various properties that you would expect from a uniform distribution.[1] For example, are there roughly an equal number of `0`s and `1`s? Does the substring `01010` occur with roughly the frequency I would expect? If I interpret the string as a series of points in the unit square $[0,1)^2$, is it true that roughly $\pi/4$ of them are within Euclidean distance 1 of the origin?

  The definition of pseudorandomness is kind of a "master" definition that encompasses all of these statistical tests and more. After all, what is a statistical test, but a polynomial-time procedure that obtains samples from a distribution and outputs a yes/no decision? Pseudorandomness means that *every* statistical test that "passes" uniform data will also "pass" pseudorandomly generated data.

5.2 Pseudorandom Generators in Practice

You are probably expecting to now see at least one example of a secure PRG. Unfortunately, things are not so simple. We have no examples of secure PRGs! If it were possible to prove

[1] For one list of such tests, see http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf.

that some function G is a secure PRG, **it would resolve the famous P vs NP problem** — the most famous unsolved problem in computer science (and arguably, all of mathematics).

The next best thing that cryptographic research can offer are **candidate PRGs**, which are *conjectured* to be secure. The best examples of such PRGs are the ones that have been subjected to significant public scrutiny and resisted all attempts at attacks so far.

In fact, the entire rest of this book is based on cryptography that is only *conjectured* to be secure. How is this possible, given the book's stated focus on *provable security?* As you progress through the book, pay attention to how all of the provable security claims are *conditional* — if X is secure then Y is secure. You will be able to trace back through this web of implications and discover that there are only a small number of underlying cryptographic primitives whose security is merely *conjectured* (PRGs are one example of such a primitive). Everything else builds on these primitives in a provably secure way.

With that disclaimer out of the way, surely *now* you can be shown an example of a conjectured secure PRG, right? There are indeed some conjectured PRGs that are simple enough to show you at this point, but you won't find them in the book. The problem is that none of these PRG candidates are really used in practice. When you really need a PRG in practice, you would actually use a PRG that is built from something called a block cipher (which we won't see until Chapter 6). A block cipher is *conceptually* more complicated than a PRG, and can even be built from a PRG (in principle). That explains why this book starts with PRGs. In practice, a block cipher is just a more useful object, so that is what you would find easily available (even implemented with specialized CPU instructions in most CPUs). When we introduce block ciphers (and pseudorandom functions), we will discuss how they can be used to construct PRGs.

How NOT to Build a PRG

We can appreciate the challenges involved in building a PRG "from scratch" by first looking at an obvious idea for a PRG and understanding why it's insecure.

Example *Let's focus on the case of a length-doubling PRG. It should take in λ bits and output 2λ bits. The output should look random when the input is sampled uniformly. A natural idea is for the candidate PRG to simply repeat the input twice. After all, if the input s is random, then $s\|s$ is also random, too, right?*

```
$G(s)$:
  return $s\|s$
```

To understand why this PRG is insecure, first let me ask you whether the following strings look like they were sampled uniformly from $\{0,1\}^8$:

$$11011101, 01010101, 01110111, 01000100, \cdots$$

Do you see any patterns? Every string has its first half equal to its second half. That is a conspicuous pattern because it is relatively rare for a uniformly chosen string to have this property.

Of course, this is exactly what is wrong with this simplistic PRG G defined above. Every output of G has equal first/second halves. But it is rare for uniformly sampled strings to have this property. We can formalize this observation as an attack against the PRG-security of G:

```
$\mathcal{A}$
$x \| y := \textsc{query}()$
return $x \overset{?}{=} y$
```

The first line means to obtain the result of QUERY *and set its first half to be the string x and its second half to be y. This calling program simply checks whether the output of* QUERY *has equal halves.*

To complete the attack, we must show that this calling program has non-negligible bias distinguishing the $\mathcal{L}_{\text{prg-}\star}$ libraries.

- *When linked to $\mathcal{L}_{\text{prg-real}}$, the calling program receives outputs of G, which always have matching first/second halves. So $\Pr[\mathcal{A} \diamond \mathcal{L}^{G}_{\text{prg-real}} \Rightarrow 1] = 1$. Below we have filled in $\mathcal{L}_{\text{prg-real}}$ with the details of our G algorithm:*

```
$\mathcal{A}$                              $\mathcal{L}^{G}_{\text{prg-real}}$
$x \| y := \textsc{query}()$   $\diamond$   $\textsc{query}()$:
return $x \overset{?}{=} y$                  $s \leftarrow \{0,1\}^\lambda$
                                              return $s \| s$
```

- *When linked to $\mathcal{L}_{\text{prg-rand}}$, the calling program receives uniform samples from $\{0,1\}^{2\lambda}$.*

```
$\mathcal{A}$                              $\mathcal{L}^{G}_{\text{prg-rand}}$
$x \| y := \textsc{query}()$   $\diamond$   $\textsc{query}()$:
return $x \overset{?}{=} y$                  $r \leftarrow \{0,1\}^{2\lambda}$
                                              return $r$
```

  $\mathcal{A}$ outputs 1 whenever we sample a string from $\{0,1\}^{2\lambda}$ with equal first/second halves. What exactly is the probability of this happening? There are several ways to see that the probability is $1/2^\lambda$ (this is like asking the probability of rolling doubles with two dice, but each die has 2^λ sides instead of 6). Therefore, $\Pr[\mathcal{A} \diamond \mathcal{L}^{G}_{\text{prg-rand}} \Rightarrow 1] = 1/2^\lambda$.

The advantage of this adversary is $1 - 1/2^\lambda$ which is certainly non-negligible — it does not even approach 0 as λ grows. This shows that G is not a secure PRG.

This example illustrates how randomness/pseudorandomness is a property of the *entire process*, not of individual strings. If you take a string of `1`s and concatenate it with another string of `1`s, you get a long string of `1`s. "Containing only `1`s" is a property of individual strings. If you take a "random string" and concatenate it with another "random string," you might not get a "random long string." Being random is not a property of an individual string, but of the entire process that generates it.

Outputs from this G have equal first/second halves, which is an obvious pattern. The challenge of desiging a secure PRG is that its outputs must have *no discernable pattern!* Any pattern will lead to an attack similar to the one shown above.

Related Concept: Random Number Generation

The security of a PRG requires the seed to be chosen uniformly. In practice, the seed has to come from somewhere. Generally a source of "randomness" is provided by the hardware or operating system, and the process that generates these random bits is (confusingly) called a random *number* generator (RNG).

In this course we won't cover low-level random *number* generation, but merely point out what makes it different than the PRGs that we study:

- The job of a PRG is to take a small amount of "ideal" (in other words, uniform) randomness and extend it.
- By contrast, an RNG usually takes many inputs over time and maintains an internal state. These inputs are often from physical/hardware sources. While these inputs are "noisy" in some sense, it is hard to imagine that they would be statistically *uniform*. So the job of the RNG is to "refine" (sometimes many) sources of noisy data into uniform outputs.

5.3 Application: Shorter Keys in One-Time-Secret Encryption

We revisit the motivating example from the beginning of this chapter. Alice & Bob share only a λ-bit key but want to encrypt a message of length $\lambda + \ell$. The main idea is to expand the key k into a longer string using a PRG G, and use the result as a one-time pad on the (longer) plaintext. More precisely, let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be a PRG, and define the following encryption scheme:

Construction 5.2 (Pseudo-OTP)

```
$\mathcal{K} = \{0,1\}^\lambda$            KeyGen:          Enc$(k,m)$:                Dec$(k,c)$:
$\mathcal{M} = \{0,1\}^{\lambda+\ell}$       $k \leftarrow \mathcal{K}$        return $G(k) \oplus m$     return $G(k) \oplus c$
$\mathcal{C} = \{0,1\}^{\lambda+\ell}$       return $k$
```

The resulting scheme will not have (perfect) one-time secrecy. That is, encryptions of m_L and m_R will not be identically distributed in general. However, the distributions will be *indistinguishable* if G is a secure PRG. The precise flavor of security obtained by this construction is the following.

Definition 5.3 *Let Σ be an encryption scheme, and let $\mathcal{L}^\Sigma_{\text{ots-L}}$ and $\mathcal{L}^\Sigma_{\text{ots-R}}$ be defined as in Definition 2.6 (and repeated below for convenience). Then Σ has* ***(computational) one-time secrecy*** *if $\mathcal{L}^\Sigma_{\text{ots-L}} \approx \mathcal{L}^\Sigma_{\text{ots-R}}$. That is, if for all polynomial-time distinguishers $\mathcal{A}$, we have* $\Pr[\mathcal{A} \diamond \mathcal{L}^\Sigma_{\text{ots-L}} \Rightarrow 1] \approx \Pr[\mathcal{A} \diamond \mathcal{L}^\Sigma_{\text{ots-R}} \Rightarrow 1]$.

```
$\mathcal{L}^\Sigma_{\text{ots-L}}$
EAVESDROP$(m_L, m_R \in \Sigma.\mathcal{M})$:
    $k \leftarrow \Sigma.\text{KeyGen}$
    $c \leftarrow \Sigma.\text{Enc}(k, m_L)$
    return $c$
```

```
$\mathcal{L}^\Sigma_{\text{ots-R}}$
EAVESDROP$(m_L, m_R \in \Sigma.\mathcal{M})$:
    $k \leftarrow \Sigma.\text{KeyGen}$
    $c \leftarrow \Sigma.\text{Enc}(k, m_R)$
    return $c$
```

This is essentially the same as Definition 2.6, except we are using $\approx$ (indistinguishability) instead of $\equiv$ (interchangeability).

Claim 5.4 *Let* pOTP *denote Construction 5.2. If* pOTP *is instantiated using a secure PRG G then* pOTP *has computational one-time secrecy.*

Proof We must show that $\mathcal{L}^{\text{pOTP}}_{\text{ots-L}} \approx \mathcal{L}^{\text{pOTP}}_{\text{ots-R}}$. As usual, we will proceed using a sequence of hybrids that begins at $\mathcal{L}^{\text{pOTP}}_{\text{ots-L}}$ and ends at $\mathcal{L}^{\text{pOTP}}_{\text{ots-R}}$. For each hybrid library, we will demonstrate that it is indistinguishable from the previous one. Note that we are allowed to use the fact that G is a secure PRG. In practical terms, this means that if we can express some hybrid library in terms of $\mathcal{L}^{G}_{\text{prg-real}}$ (one of the libraries in the PRG security definition), we can replace it with its counterpart $\mathcal{L}^{G}_{\text{prg-rand}}$ (or vice-versa). The PRG security of G says that such a change will be indistinguishable.

$\mathcal{L}^{\text{pOTP}}_{\text{ots-L}}$:

```
L^pOTP_ots-L
EAVESDROP($m_L, m_R \in \{0,1\}^{\lambda+\ell}$):
    $k \leftarrow \{0,1\}^\lambda$
    $c := G(k) \oplus m_L$
    return $c$
```

The starting point is $\mathcal{L}^{\text{pOTP}}_{\text{ots-L}}$, shown here with the details of pOTP filled in.

$\mathcal{L}_{\text{hyb-1}}$:

```
EAVESDROP($m_L, m_R$):
    $z \leftarrow$ QUERY()
    $c := z \oplus m_L$
    return $c$
```

$\diamond$

```
L^G_prg-real
QUERY():
    $s \leftarrow \{0,1\}^\lambda$
    return $G(s)$
```

The first hybrid step is to factor out the computations involving G, in terms of the $\mathcal{L}^{G}_{\text{prg-real}}$ library.

$\mathcal{L}_{\text{hyb-2}}$:

```
EAVESDROP($m_L, m_R$):
    $z \leftarrow$ QUERY()
    $c := z \oplus m_L$
    return $c$
```

$\diamond$

```
L^G_prg-rand
QUERY():
    $r \leftarrow \{0,1\}^{\lambda+\ell}$
    return $r$
```

From the PRG security of G, we may replace the instance of $\mathcal{L}^{G}_{\text{prg-real}}$ with $\mathcal{L}^{G}_{\text{prg-rand}}$. The resulting hybrid library $\mathcal{L}_{\text{hyb-2}}$ is indistinguishable from the previous one.

$\mathcal{L}_{\text{hyb-3}}$:

```
L^OTP_ots-L
EAVESDROP($m_L, m_R$):
    $z \leftarrow \{0,1\}^{\lambda+\ell}$
    $c := z \oplus m_L$
    return $c$
```

A subroutine has been inlined. Note that the resulting library is precisely $\mathcal{L}^{\text{OTP}}_{\text{ots-L}}$ involving **standard one-time pad** on plaintexts of size $\lambda + \ell$. We have essentially proven that pOTP is indistinguishable from standard OTP, and therefore we can apply the security of OTP.

$\mathcal{L}_{\text{hyb-4}}$:

```
$\mathcal{L}^{\text{OTP}}_{\text{ots-R}}$
EAVESDROP($m_L, m_R$):
    $z \leftarrow \{0,1\}^{\lambda+\ell}$
    $c := z \oplus m_R$
    return $c$
```

The (perfect) one-time secrecy of rOTP allows us to replace $\mathcal{L}^{\text{OTP}}_{\text{ots-L}}$ with $\mathcal{L}^{\text{OTP}}_{\text{ots-R}}$; they are interchangeable.

The rest of the proof is essentially a "mirror image" of the previous steps, in which we perform the same steps but in reverse (and with m_R being used instead of m_L).

$\mathcal{L}_{\text{hyb-5}}$:

```
EAVESDROP($m_L, m_R$):
    $z \leftarrow$ QUERY()
    $c := z \oplus m_R$
    return $c$
```

$\diamond$

```
$\mathcal{L}^{G}_{\text{prg-rand}}$
QUERY():
    $r \leftarrow \{0,1\}^{\lambda+\ell}$
    return $r$
```

A statement has been factored out into a subroutine, which happens to exactly match $\mathcal{L}^{G}_{\text{prg-rand}}$.

$\mathcal{L}_{\text{hyb-6}}$:

```
EAVESDROP($m_L, m_R$):
    $z \leftarrow$ QUERY()
    $c := z \oplus m_R$
    return $c$
```

$\diamond$

```
$\mathcal{L}^{G}_{\text{prg-real}}$
QUERY():
    $s \leftarrow \{0,1\}^{\lambda}$
    return $G(s)$
```

From the PRG security of G, we can replace $\mathcal{L}^{G}_{\text{prg-rand}}$ with $\mathcal{L}^{G}_{\text{prg-real}}$. The resulting library is indistinguishable from the previous one.

$\mathcal{L}^{\text{pOTP}}_{\text{ots-R}}$:

```
$\mathcal{L}^{\text{pOTP}}_{\text{ots-R}}$
EAVESDROP($m_L, m_R$):
    $k \leftarrow \{0,1\}^{\lambda}$
    $c := G(k) \oplus m_R$
    return $c$
```

A subroutine has been inlined. The result is $\mathcal{L}^{\text{pOTP}}_{\text{ots-R}}$.

Summarizing, we showed a sequence of hybrid libraries satisfying the following:

$$\mathcal{L}^{\text{pOTP}}_{\text{ots-L}} \equiv \mathcal{L}_{\text{hyb-1}} \approx \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}_{\text{hyb-5}} \approx \mathcal{L}_{\text{hyb-6}} \equiv \mathcal{L}^{\text{pOTP}}_{\text{ots-R}}.$$

Hence, $\mathcal{L}^{\text{pOTP}}_{\text{ots-L}} \approx \mathcal{L}^{\text{pOTP}}_{\text{ots-R}}$, and pOTP has (computational) one-time secrecy. ■

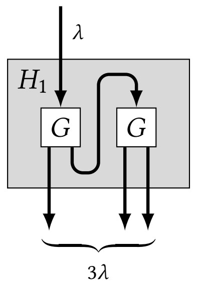
5.4 Extending the Stretch of a PRG

The *stretch* of a PRG measures how much longer its output is than its input. Can we use a PRG with small stretch to construct a PRG with larger stretch? The answer is yes, but only if you do it the right way!

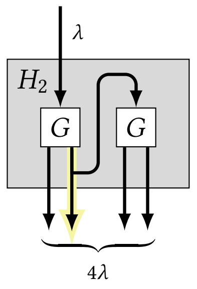
Two Approaches to Increase Stretch

Suppose $G : \{0,1\}^{\lambda} \rightarrow \{0,1\}^{2\lambda}$ is a length-doubling PRG (*i.e.*, a PRG with stretch λ). Below are two ideas for constructing a PRG with longer stretch:

```
$H_1(s)$:
    $x \| y := G(s)$
    $u \| v := G(y)$
    return $x \| u \| v$
```



```
$H_2(s)$:
    $x \| y := G(s)$
    $u \| v := G(y)$
    return $x \| y \| u \| v$
```



Although the constructions are similar, only one of them is secure. Before reading any further, can you guess which of H_1, H_2 is a secure PRG and which is insecure? By carefully comparing these two approaches, I hope you develop a better understanding of the PRG security definition.

A Security Proof

I think it's helpful to illustrate the "stragey" of security proofs by starting from the desired conclusion and working backwards. What better way to do this than as a Socratic dialogue in the style of Galileo?[2]

Salviati: *I'm sure that H_1 is the secure PRG.*

Simplicio: If I understand the security definition for PRGs correctly, you mean that the output of H_1 looks indistinguishable from uniform, when the input to H_1 is uniform. Why do you say that?

Salviati: *Simple! H_1's output consists of segments called x, u, and v. Each of these are outputs of G, and since G itself is a PRG its outputs look uniform.*

Simplicio: I wish I had your boldness, Salviati. I myself am more cautious. If G is a secure PRG, then its outputs are indeed indistinguishable from uniform, but surely *only when its input is uniform!* Are you so sure that's the case here?

Salviati: *You raise a good point, Simplicio. In these endeavors it is always preferable to err on the side of caution. When we want to claim that H_1 is a secure PRG, we consider the nature of its outputs when its seed s is uniform. Since H_1 sends that seed s directly into G, your concern is addressed.*

Simplicio: Yes, I can see how in the expression $x \| y := G(s)$ the input to G is uniform, and so its outputs x and y are indistinguishable from random. Since x is part of H_1's output, we are making progress towards showing that the entire output of H_1 is indistinguishable from random! However, the output of H_1 also contains terms u and v. When I examine how they are generated, as $u \| v := G(y)$, I become concerned again. Surely y is not uniform, so I see no way to apply the security if G!

[2]Don't answer that.

Salviati: *Oh, bless your heart. The answer could not be any more obvious! It is true that y is not uniformly distributed. But did you not just convince yourself that y is* indistinguishable *from uniform? Should that suffice?*

Simplicio: Incredible! I believe I understand now. Let me try to summarize: We suppose the input s to H_1 is chosen uniformly, and examine what happens to H_1's outputs. In the expression $x\|y := G(s)$, the input to G is uniform, and thus x and y are indistinguishable from uniform. Now, considering the expression $u\|v := G(y)$, the result is indistinguishable from a scenario in which y is truly uniform. But if y were truly uniform, those outputs u and v would be indistinguishable from uniform! Altogether, x, u, and v (the outputs of H_1) are each indistinguishable from uniform!

I hope that was as fun for you as it was for me.[3] The formal security proof and its sequence of hybrids will follow the outline given in Simplicio's summary. We start by applying the PRG security definition to the first call to G, and replace its outputs with truly uniform values. After this change, the input to the second call to G becomes uniform, allowing us to apply the PRG security definition again.

Claim 5.5 *If G is a secure length-doubling PRG, then H_1 (defined above) is a secure (length-tripling) PRG.*

Proof One of the trickier aspects of this proof is that we are using a secure PRG G to prove the security of another PRG H_1. That means both $\mathcal{L}^{H_1}_{\text{prg-}\star}$ and $\mathcal{L}^{G}_{\text{prg-}\star}$ will appear in this proof. Both libraries/interfaces have a subroutine named "query", and we will rename these subroutines query_{H_1} and query_G to disambiguate.

We want to show that $\mathcal{L}^{H_1}_{\text{prg-real}} \approx \mathcal{L}^{H_1}_{\text{prg-rand}}$. As usual, we do so with a hybrid sequence. Since we assume that G is a secure PRG, we are allowed to use the fact that $\mathcal{L}^{G}_{\text{prg-real}} \approx \mathcal{L}^{G}_{\text{prg-rand}}$.

$\mathcal{L}^{H}_{\text{prg-real}}$:

$\mathcal{L}^{H_1}_{\text{prg-real}}$
```
query_{H_1}():
    $s \leftarrow \{0,1\}^\lambda$
    $x\|y := G(s)$       } $H_1(s)$
    $u\|v := G(y)$
    return $x\|u\|v$
```

The starting point is $\mathcal{L}^{H_1}_{\text{prg-real}}$, shown here with the details of H_1 filled in.

```
query_{H_1}():
    $x\|y := \text{query}_G()$
    $u\|v := G(y)$
    return $x\|u\|v$
```
$\diamond$

$\mathcal{L}^{G}_{\text{prg-real}}$
```
query_G():
    $s \leftarrow \{0,1\}^\lambda$
    return $G(s)$
```

The first invocation of G has been factored out into a subroutine. The resulting hybrid library includes an instance of $\mathcal{L}^{G}_{\text{prg-real}}$.

[3] If you're wondering what the hell just happened: In Galileo's 1632 book *Dialogue Concerning the Two Chief World Systems*, he lays out the arguments for heliocentrism using a dialog between Salviati (who advocated the heliocentric model) and Simplicio (who believed the geocentric model).

```
QUERY_{H_1}():
  $x \| y$ := QUERY_G()
  $u \| v := G(y)$
  return $x \| u \| v$
```
$\diamond$
```
$\mathcal{L}^G_{\text{prg-rand}}$
QUERY_G():
  $r \leftarrow \{0,1\}^{2\lambda}$
  return $r$
```

From the PRG security of G, we can replace the instance of $\mathcal{L}^G_{\text{prg-real}}$ with $\mathcal{L}^G_{\text{prg-rand}}$. The resulting hybrid library is indistinguishable.

```
QUERY_{H_1}():
  $x \| y \leftarrow \{0,1\}^{2\lambda}$
  $u \| v := G(y)$
  return $x \| u \| v$
```

A subroutine has been inlined.

```
QUERY_{H_1}():
  $x \leftarrow \{0,1\}^{\lambda}$
  $y \leftarrow \{0,1\}^{\lambda}$
  $u \| v := G(y)$
  return $x \| u \| v$
```

Choosing 2λ uniformly random bits and then splitting them into two halves has exactly the same effect as choosing λ uniformly random bits and independently choosing λ more.

```
QUERY_{H_1}():
  $x \leftarrow \{0,1\}^{\lambda}$
  $u \| v$ := QUERY_G()
  return $x \| u \| v$
```
$\diamond$
```
$\mathcal{L}^G_{\text{prg-real}}$
QUERY_G():
  $s \leftarrow \{0,1\}^{\lambda}$
  return $G(s)$
```

The remaining appearance of G has been factored out into a subroutine. Now $\mathcal{L}^G_{\text{prg-real}}$ makes its second appearance.

```
QUERY_{H_1}():
  $x \leftarrow \{0,1\}^{\lambda}$
  $u \| v$ := QUERY_G()
  return $x \| u \| v$
```
$\diamond$
```
$\mathcal{L}^G_{\text{prg-rand}}$
QUERY_G():
  $r \leftarrow \{0,1\}^{2\lambda}$
  return $r$
```

Again, the PRG security of G lets us replace $\mathcal{L}^G_{\text{prg-real}}$ with $\mathcal{L}^G_{\text{prg-rand}}$. The resulting hybrid library is indistinguishable.

```
QUERY_{H_1}():
  $x \leftarrow \{0,1\}^{\lambda}$
  $u \| v \leftarrow \{0,1\}^{2\lambda}$
  return $x \| u \| v$
```

A subroutine has been inlined.

$\mathcal{L}^{H_1}_{\text{prg-rand}}$:
```
$\mathcal{L}^{H_1}_{\text{prg-rand}}$
QUERY_{H_1}():
  $r \leftarrow \{0,1\}^{3\lambda}$
  return $r$
```

Similar to above, concatenating λ uniform bits with 2λ independently uniform bits has the same effect as sampling 3λ uniform bits. The result of this change is $\mathcal{L}^{H_1}_{\text{prg-rand}}$.

Through this sequence of hybrid libraries, we showed that:

$$\mathcal{L}^{H_1}_{\text{prg-real}} \equiv \mathcal{L}_{\text{hyb-1}} \approx \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}_{\text{hyb-5}} \approx \mathcal{L}_{\text{hyb-6}} \equiv \mathcal{L}_{\text{hyb-7}} \equiv \mathcal{L}^{H_1}_{\text{prg-rand}}.$$

Hence, H_1 is a secure PRG. ■

Where the Proof Breaks Down for H_2

The only difference between H_1 and H_2 is that the variable y is included in the output. How does that minor change affect the reasoning that we applied to H_1?

```
$H_2(s)$:
  $x \| y := G(s)$
  $u \| v := G(y)$
  return $x \| y \| u \| v$
```

We argued that outputs u and v are indistinguishable from uniform since its input y is also indistinguishable from random. But it's not quite so simple: A PRG's output is indistinguishable from random if (1) its seed is uniform, and (2) *the seed is not used for anything else!* This construction H_2 violates condition (2) because it includes the "seed" y in the output.

We can see this idea reflected in the formal PRG definition. In $\mathcal{L}_{\text{prg-real}}$, the seed s is chosen uniformly, given as input to G, and then *goes out of scope!* If we try to reproduce the security proof for H_1 with H_2 instead, we'll get stuck when we are trying to factor out the second call to G in terms of $\mathcal{L}_{\text{prg-real}}$:

```
QUERY$_{H_2}$():
  $x \leftarrow \{0,1\}^\lambda$
  $y \leftarrow \{0,1\}^\lambda$
  $u \| v := G(y)$
  return $x \| y \| u \| v$
```

$\rightsquigarrow$

```
QUERY$_{H_1}$():                 $\diamond$   $\mathcal{L}^G_{\text{prg-real}}$
  $x \leftarrow \{0,1\}^\lambda$                QUERY$_G$():
  $u \| v :=$ QUERY$_G$()                $s \leftarrow \{0,1\}^\lambda$
  return $x \| y \| u \| v$             return $G(s)$
```

scope error! y undefined

We are trying to factor out the two highlighted lines into a separate library, renaming y into s in the process. But s can only exist inside the private scope of the new library, while there still exists a "dangling reference" y in the original library.

Speaking more generally about PRGs, suppose we have a call to G somewhere and want to argue that its outputs are pseudorandom. We can only express this call to G in terms of $\mathcal{L}^G_{\text{prg-real}}$ if the input to G is uniform and is used nowhere else. That's not true here – we can't express one of the calls to G in terms of $\mathcal{L}^G_{\text{prg-real}}$, so we can't be sure that the outputs of that call to G look random.

These subtle issues are not limited to PRGs. Every hybrid security proof in this course includes steps where we factor out some statements in terms of some pre-existing library. Don't take these steps for granted! They will fail (often because of scoping issues) if the construction isn't using the building block correctly. You should always treat such "factoring out" steps as "sanity checks" for your proof.

A Concrete Attack on H_2

So far, we've only demonstrated that we get stuck when trying to prove the security of H_2. But that doesn't necessarily mean that H_2 is insecure – it could mean that we're just not clever enough to see a different security proof. To show that H_2 is actually insecure, we must demonstrate a successful distinguishing attack.

Attacking a PRG amounts to finding "patterns" in their outputs. Does H_2 have a pattern in its outputs? Yes, in this case the pattern is that if you write the output in the form $x\|y\|u\|v$, then $u\|v$ is always equal to $G(y)$. The calling program can check for this condition, which is unlikely to happen for truly uniform values.

You may wonder, is it legal for the calling program to compute $G(y)$? Well, G is a publicly known algorithm (Kerckhoffs' principle!), and y is right there as part of the input. Nothing prevents the calling program from running G "in its head."[4]

Claim 5.6 *Construction H_2 is **not** a secure PRG, even if G is.*

Proof Consider the following distinguisher $\mathcal{A}$:

```
$x\|y\|u\|v := \textsc{query}()$
return $G(y) \overset{?}{=} u\|v$
```

When $\mathcal{A}$ is linked to $\mathcal{L}^{H_2}_{\text{prg-real}}$, the outputs indeed satisfy the condition $G(y) = u\|v$, so $\mathcal{A}$ outputs `true` with probability 1.

When $\mathcal{A}$ is linked to $\mathcal{L}^{H_2}_{\text{prg-rand}}$, the outputs are truly uniform. It is helpful to imagine x and y being chosen before u and v. As soon as y is chosen, the value $G(y)$ is uniquely determined, since G is a deterministic algorithm. Then $\mathcal{A}$ will output `true` if $u\|v$ is chosen exactly to equal this $G(y)$. Since u and v are chosen uniformly, and are a total of 2κ bits long, this event happens with probability $1/2^{2\kappa}$.

$\mathcal{A}$'s advantage is the difference in these probabilities: $1 - 1/2^{2\kappa}$, which is non-negligible. ■

Discussion

In the attack on H_2, we never tried to distinguish the output of G from uniform. H_2 is insecure even if G is the best PRG in the world! It's insecure because of the incorrect way it *uses* G.

From now on in this book, we'll be studying higher-level constructions that are assembled from various building blocks — in this chapter, fancy PRGs constructed from simpler PRGs. "Security" means: if the building blocks are secure then the construction is secure. "Insecurity" means: *even if the building blocks are secure,* the construction can be insecure. So when you're showing insecurity, you shouldn't directly attack the building blocks! You should assume the building blocks are secure and attack *the way that the building blocks are being used.*

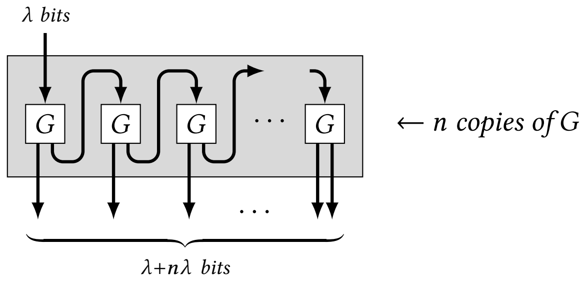
[4] Compare to the case of distinguishing $G(s)$ from uniform, for a secure G. The calling program knows the algorithm G but doesn't have the seed s — it only knows the *output* $G(s)$. In the case of H_2, the calling program learns both y and $G(y)$!

⋆ 5.5 Applications: Stream Cipher & Symmetric Ratchet

The PRG-feedback construction can be generalized in a natural way, by continuing to feed part of G's output into G again. The proof works in the same way as for the previous construction — the security of G is applied one at a time to each application of G.

Claim 5.7 *If G is a secure length-doubling PRG, then for any n (polynomial function of λ) the following construction H_n is a secure PRG with stretch $n\lambda$:*

```
$\underline{H_n(s)}$:
  $s_0 := s$
  for $i = 1$ to $n$:
    $s_i \| t_i := G(s_{i-1})$
  return $t_1 \| \cdots \| t_n \| s_n$
```



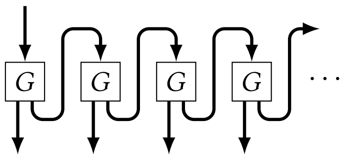
The fact that this chain of PRGs can be extended indefinitely gives another useful functionality:

Definition 5.8 (Stream cipher) *A **stream cipher** is an algorithm G that takes a seed s and length ℓ as input, and outputs a string. It should satisfy the following requirements:*

1. *$G(s,\ell)$ is a string of length ℓ.*
2. *If $i < j$, then $G(s,i)$ is a prefix of $G(s,j)$.*
3. *For each n, the function $G(\cdot,n)$ is a secure PRG.*

Because of the 2nd rule, you might want to think about a single infinitely long string that is the *limit* of $G(s,n)$ as n goes to infinity. The finite-length strings $G(s,n)$ are all the prefixes of this infinitely long string.

The PRG-feedback construction can be used to construct a secure stream cipher in the natural way: given seed s and length ℓ, keep iterating the PRG-feedback main loop until ℓ bits have been generated.



Symmetric Ratchet

Suppose Alice & Bob share a symmetric key k and are using a secure messaging app to exchange messages over a long period of time. Later in the course we will see techniques that Alice & Bob could use to securely encrypt many messages using a single key. However, suppose Bob's device is compromised and an attacker learns k. Then the attacker can decrypt all past, present, and future ciphertexts that it saw!

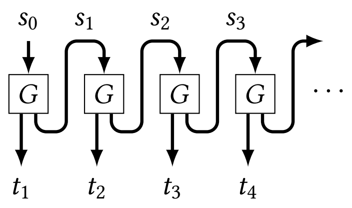
Alice & Bob can protect against such a key compromise by using the PRG-feedback stream cipher to constantly "update" their shared key. Suppose they do the following, starting with their shared key k:

- They use k to seed a chain of length-doubling PRGs, and both obtain the same stream of pseudorandom keys $t_1, t_2, \ldots$.
- They use t_i as a key to send/receive the ith message. The details of the encryption are not relevant to this example.
- After making a call to the PRG, they erase the PRG input from memory, and only remember the PRG's output. After using t_i to send/receive a message, they also erase it from memory.

This way of using and forgetting a sequence of keys is called a **symmetric ratchet**.

Construction 5.9 (Symm Ratchet)

```
$s_0 = k$
for $i = 1$ to $\infty$:
  $s_i \| t_i := G(s_{i-1})$
  **erase** $s_{i-1}$ from memory
  use $t_i$ to encrypt/decrypt the $i$th message
  **erase** $t_i$ from memory
```



Suppose that an attacker compromises Bob's device after n ciphertexts have been sent. The only value residing in memory is s_n, which the attacker learns. Since G is deterministic, the attacker can now compute $t_{n+1}, t_{n+2}, \ldots$ in the usual way and decrypt all future ciphertexts that are sent.

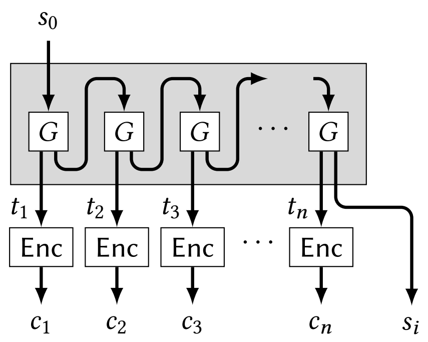
However, we can show that the attacker learns no information about $t_1, \ldots, t_n$ from s_n, which implies that the previous ciphertexts remain safe. By compromising the key s_n, the adversary only compromises the security of *future* messages, but not *past* messages. Sometimes this property is called **forward secrecy**, meaning that messages in the present are protected against a key-compromise that happens in the future.

This construction is called a **ratchet**, since it is easy to advance the key sequence in the forward direction (from s_n to s_{n+1}) but hard to reverse it (from s_{n+1} to s_n). The exercises explore the problem of explicitly reversing the ratchet, but the more relevant property for us is whether the attacker learns anything about the ciphertexts that were generated before the compromise.

Claim 5.10 *If the symmetric ratchet (Construction 5.9) is used with a secure PRG G and an encryption scheme Σ that has uniform ciphertexts (and $\Sigma.\mathcal{K} = \{0,1\}^\lambda$), then the first n ciphertexts are pseudorandom, even to an eavesdropper who compromises the key s_n.*

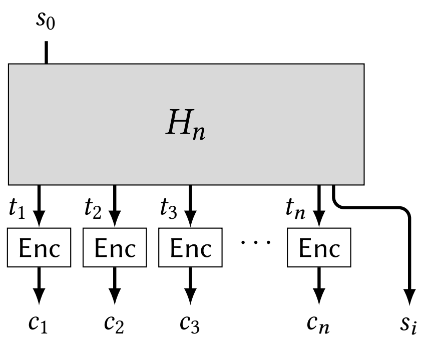
Proof We are considering an attack scenario in which n plaintexts are encrypted, and the adversary sees their ciphertexts as well as the ratchet-key s_n. This situation is captured by the following library:

```
ATTACK($m_1, \ldots, m_n$):
  $s_0 \leftarrow \{0,1\}^\lambda$
  for $i = 1$ to $n$:
    $s_i \| t_i := G(s_{i-1})$
    $c_i \leftarrow \Sigma.\text{Enc}(t_i, m_i)$
  return $(c_1, \ldots, c_n, s_n)$
```



As we have seen, the shaded box (the process that computes $t_1, \ldots, t_n$ from s_0) is actually a PRG. Let us rewrite the library in terms of this PRG H_n:

```
ATTACK($m_1, \ldots, m_n$):
  $s_0 \leftarrow \{0,1\}^\lambda$
  $t_1 \| \cdots \| t_n \| s_n := H_n(s_0)$
  for $i = 1$ to $n$:
    $c_i \leftarrow \Sigma.\text{Enc}(t_i, m_i)$
  return $(c_1, \ldots, c_n, s_n)$
```



Now, we can apply the PRG security of H_n and instead choose $t_1, \ldots, t_n$ and s_n uniformly. This change is indistinguishable, by the security of the PRG. Note that we have not written out the standard explicit steps (factor out the first two lines of ATTACK in terms of $\mathcal{L}_{\text{prg-real}}$, replace with $\mathcal{L}_{\text{prg-rand}}$, and inline).

```
ATTACK($m_1, \ldots, m_n$):
  for $i = 1$ to $n$:
    $t_i \leftarrow \{0,1\}^\lambda$
  $s_n \leftarrow \{0,1\}^\lambda$
  for $i = 1$ to $n$:
    $c_i \leftarrow \Sigma.\text{Enc}(t_i, m_i)$
  return $(c_1, \ldots, c_n, s_n)$
```

$\equiv$

```
ATTACK($m_1, \ldots, m_n$):
  for $i = 1$ to $n$:
    $t_i \leftarrow \{0,1\}^\lambda$
    $c_i \leftarrow \Sigma.\text{Enc}(t_i, m_i)$
  $s_n \leftarrow \{0,1\}^\lambda$
  return $(c_1, \ldots, c_n, s_n)$
```

At this point, the encryption scheme is being used "as intended," meaning that we generate its keys t_i uniformly/indepenendtly, and use each key only for one encryption and nothing else. Formally speaking, this means we can factor out the body of the for-loop in terms of $\mathcal{L}_{\text{ots\$-real}}$:

```
ATTACK($m_1, \ldots, m_n$):
  for $i = 1$ to $n$:
    $c_i \leftarrow$ CTXT($m_i$)
  $s_n \leftarrow \{0,1\}^\lambda$
  return $(c_1, \ldots, c_n, s_n)$
```

$\diamond$

```
$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$
CTXT($m \in \Sigma.\mathcal{M}$):
  $k \leftarrow \Sigma.\text{KeyGen}$
  $c \leftarrow \Sigma.\text{Enc}(k, m)$
  return $c$
```

We can now replace $\mathcal{L}_{\text{ots\$-real}}$ with $\mathcal{L}_{\text{ots\$-rand}}$ and inline the subroutine (without showing the intermediate library). The result is:

```
ATTACK($m_1, \ldots, m_n$):
  for $i = 1$ to $n$:
    $c_i \leftarrow \Sigma.\mathcal{C}$
  $s_n \leftarrow \{0,1\}^\lambda$
  return $(c_1, \ldots, c_n, s_n)$
```

This final library is indistinguishable from the first one. As promised, we showed that the attacker cannot distinguish the first n ciphertexts from random values, even when seeing s_n. ■

This proof used the uniform-ciphertexts property, but the same logic applies to basically any encryption property you care about — just imagine factoring out the encryption steps in terms of a different library than $\mathcal{L}_{\text{ots\$-real}}$.

Exercises

5.1. Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be an injective (*i.e.*, 1-to-1) PRG. Consider the following distinguisher:

```
$\mathcal{A}$
$x :=$ QUERY()
for all $s' \in \{0,1\}^\lambda$:
  if $G(s') = x$ then return 1
return 0
```

(a) What is the advantage of $\mathcal{A}$ in distinguishing $\mathcal{L}^G_{\text{prg-real}}$ and $\mathcal{L}^G_{\text{prg-rand}}$? Is it negligible?

(b) Does this contradict the fact that G is a PRG? Why or why not?

(c) What happens to the advantage if G is not injective?

5.2. Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be an injective PRG, and consider the following distinguisher:

```
$\mathcal{A}$
$x :=$ QUERY()
$s' \leftarrow \{0,1\}^\lambda$
return $G(s') \overset{?}{=} x$
```

What is the advantage of $\mathcal{A}$ in distinguishing $\mathcal{L}^G_{\text{prg-real}}$ from $\mathcal{L}^G_{\text{prg-rand}}$?

Hint: When computing $\Pr[\mathcal{A} \diamond \mathcal{L}^G_{\text{prg-rand}}$ outputs 1], separate the probabilities based on whether x is a possible output of G or not.

5.3. For any PRG $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ there will be many strings in $\{0,1\}^{\lambda+\ell}$ that are impossible to get as output of G. Let S be any such set of impossible G-outputs, and consider the following adversary that has S hard-coded:

```
𝒜
$x :=$ QUERY()
return $x \overset{?}{\in} S$
```

What is the advantage of $\mathcal{A}$ in distinguishing $\mathcal{L}^G_{\text{prg-real}}$ from $\mathcal{L}^G_{\text{prg-rand}}$? Why does an adversary like this one not automatically break every PRG?

5.4. Show that the scheme from Section 5.3 does not have *perfect* one-time secrecy, by showing that there must exist two messages m_1 and m_2 whose ciphertext distributions differ.

Hint: There must exist strings $s_1, s_2 \in \{0,1\}^{\lambda+\ell}$ where $s_1 \in \text{im}(G)$, and $s_2 \notin \text{im}(G)$. Use these two strings to find two messages m_1 and m_2 whose ciphertext distributions assign different probabilities to s_1 and s_2. Note that it is legitimate for an attacker to "know" s_1 and s_2, as these are properties of G alone, and do not depend on the random choices made "at runtime" — when the library executes the encryption algorithm.

5.5. The proof of Claim 5.5 applies the PRG security rule to both of the calls to G, starting with the first one. Describe what happens when you try to apply the PRG security of G to these two calls in the opposite order. Does the proof still work, or does it work only in the order that was presented?

5.6. Let $\ell' > \ell > 0$. Extend the "PRG feedback" construction to transform any PRG of stretch ℓ into a PRG of stretch ℓ'. Formally define the new PRG and prove its security using the security of the underlying PRG.

5.7. Prove that if G is a secure PRG, then so is the function $H(s) = G(\overline{s})$.

5.8. Let $G : \{0,1\}^\lambda \to \{0,1\}^{3\lambda}$ be a secure length-**tripling** PRG. For each function below, state whether it is also a secure PRG. If the function is a secure PRG, give a proof. If not, then describe a successful distinguisher and explicitly compute its advantage. When we write $a\|b\|c := G(s)$, each of a, b, c have length λ.

(a)
```
$H(s)$:
  $x\|y\|z := G(s)$
  return $G(x)\|G(z)$
```

(b)
```
$H(s)$:
  $x\|y\|z := G(s)$
  return $x\|y$
```

(c)
```
$H(s)$:
  $x := G(s)$
  $y := G(s)$
  return $x\|y$
```

(d)
```
$H(s)$:
  $x := G(s)$
  $y := G(0^\lambda)$
  return $x\|y$
```

(e)
```
$H(s)$:
  $x := G(s)$
  $y := G(0^\lambda)$
  return $x \oplus y$
```

(f)
```
// $H : \{0,1\}^{2\lambda} \to \{0,1\}^{3\lambda}$

$H(s_L \| s_R)$:
  $x := G(s_L)$
  $y := G(s_R)$
  return $x \oplus y$
```

(g)
```
// $H : \{0,1\}^{2\lambda} \to \{0,1\}^{6\lambda}$

$H(s_L \| s_R)$:
  $x := G(s_L)$
  $y := G(s_R)$
  return $x \| y$
```

5.9. Let $G : \{0,1\}^{\lambda} \to \{0,1\}^{3\lambda}$ be a secure length-**tripling** PRG. Prove that each of the following functions is also a secure PRG:

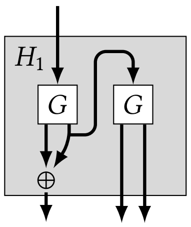
(a)
```
// $H : \{0,1\}^{2\lambda} \to \{0,1\}^{4\lambda}$

$H(s_L \| s_R)$:
  $y := G(s_R)$
  return $s_L \| y$
```

Note that H includes half of its input directly in the output. How do you reconcile this fact with the conclusion of Exercise 5.14(b)?

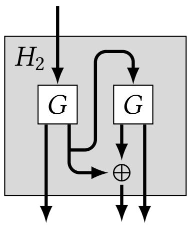
(b)
```
// $H : \{0,1\}^{2\lambda} \to \{0,1\}^{3\lambda}$

$H(s_L \| s_R)$:
  return $G(s_L)$
```

★ 5.10. Let G be a secure length-doubling PRG. One of the following constructions is a secure PRG and one is not. Which is which? Give a security proof for one and an attack for the other.

```
$H_1(s)$:
  $x \| y := G(s)$
  $u \| v := G(y)$
  return $(x \oplus y) \| u \| v$
```



```
$H_2(s)$:
  $x \| y := G(s)$
  $u \| v := G(y)$
  return $x \| (y \oplus u) \| v$
```



Hint: Usually when something is insecure, it's insecure for *any* choice of building block. In this case, the attack only works for certain G. Basically, you will need to construct a particular G, prove that it's a secure PRG, and then prove that H_1/H_2 is not secure when using this G.

5.11. A frequently asked question in cryptography forums is whether it's possible to determine which PRG implementation was used by looking at output samples.

Let G_1 and G_2 be two PRGs with matching input/output lengths. Define two libraries $\mathcal{L}^{G_1}_{\text{which-prg}}$ and $\mathcal{L}^{G_2}_{\text{which-prg}}$ as follows:

```
$\mathcal{L}^{G_1}_{\text{which-prg}}$

QUERY():
  $s \leftarrow \{0,1\}^{\lambda}$
  return $G_1(s)$
```

```
$\mathcal{L}^{G_2}_{\text{which-prg}}$

QUERY():
  $s \leftarrow \{0,1\}^{\lambda}$
  return $G_2(s)$
```

Prove that if G_1 and G_2 are both secure PRGs, then $\mathcal{L}^{G_1}_{\text{which-prg}} \approx \mathcal{L}^{G_2}_{\text{which-prg}}$ — that is, it is infeasible to distinguish which PRG was used simply by receiving output samples.

5.12. Let G_1 and G_2 be deterministic functions, each accepting inputs of length λ and producing outputs of length 3λ.

(a) Define the function $H(s_1 \| s_2) = G_1(s_1) \oplus G_2(s_2)$. Prove that if **either** of G_1 or G_2 (or both) is a secure PRG, then so is H.

(b) What can you say about the simpler construction $H(s) = G_1(s) \oplus G_2(s)$, when one of G_1, G_2 is a secure PRG?

★ 5.13. Prove that if PRGs exist, then P ≠ NP.

Hint: $\{y \mid \exists s : G(s) = y\} \in$ NP. Prove the contrapositive! Use the powerful assumption that P = NP to construct an efficient adversary to attack any candidate PRG.

5.14. (a) Let f be any function. Show that the following function G is **not** a secure PRG, no matter what f is. Describe a successful distinguisher and explicitly compute its advantage:

```
G(s):
  return $s \| f(s)$
```

(b) Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be a candidate PRG. Suppose there is a polynomial-time algorithm V with the property that it inverts G with non-negligible probability. That is,

$$\Pr_{s \leftarrow \{0,1\}^\lambda} \big[V(G(s)) = s \big] \text{ is non-negligible.}$$

Show that if an algorithm V exists with this property, then G is not a secure PRG. In other words, construct a distinguisher contradicting the PRG-security of G and show that it achieves non-negligible distinguishing advantage.

Note: Don't assume anything about the output of V other than the property shown above. In particular, V might very frequently output the "wrong" thing.

5.15. Let $s_0, s_1, \ldots$ and $t_1, t_2, \ldots$ be defined as in the symmetric ratchet (Construction 5.9).

(a) Prove that if G is a secure PRG then the following two libraries are indistinguishable, for any polynomial-time algorithm $\mathcal{A}$:

```
$\mathcal{L}_{\text{left}}$
TEST():
  $s_{n-1} \leftarrow \{0,1\}^\lambda$
  $s_n \| t_n := G(s_{n-1})$
  $\tilde{t} = \mathcal{A}(s_n)$
  return $\tilde{t} \overset{?}{=} t_n$
```

```
$\mathcal{L}_{\text{right}}$
TEST():
  $s_n \leftarrow \{0,1\}^\lambda$
  $\tilde{t} = \mathcal{A}(s_n)$
  $t_n \leftarrow \{0,1\}^\lambda$
  return $\tilde{t} \overset{?}{=} t_n$
```

(b) What is Pr[TEST outputs true] in $\mathcal{L}_{\text{right}}$?

(c) Prove that for any polynomial-time algorithm $\mathcal{A}$, $\Pr[\mathcal{A}(s_n) = t_n]$ is negligible, where s_n, t_n are generated as in the symmetric ratchet construction.

(d) Prove that for any polynomial-time algorithm $\mathcal{A}$, $\Pr[\mathcal{A}(s_n) = s_{n-1}]$ is negligible. In other words, "turning the ratchet backwards" is a hard problem.

Hint: the proof should be a few lines, a direct corollary of part (c).