

An architecture for robust pseudo-random generation and applications to `/dev/random`

Boaz Barak
Institute for Advanced Study
boaz@ias.edu

Shai Halevi
IBM
shaih@alum.mit.edu

April 11, 2005

Abstract

We present a formal model and a simple architecture for robust pseudorandom generation that ensures resilience in the face of an observer with partial knowledge (or even partial control) of the generator's entropy source. Our model and architecture ensure the following properties:

- *Resilience.* The generator's output looks random to an observer with no knowledges of the internal state. This holds even if that observer has complete control over data that is used to refresh the internal state.
- *Forward security.* Past output of the generator looks random to an observer with no knowledges of the past internal state, even if it was later able to learn the internal state.
- *Backward security/Break-in recovery.* Future output of the generator looks random, even to an observer with knowledge of the current state, provided that the generator is refreshed with data of sufficient entropy.

Architectures such as above were suggested before. This work differs from previous attempts in that we present a formal model for robust pseudo-random generation, and provide a formal proof within this model for the security of our architecture. To our knowledge, this is the first attempt at a rigorous model for this problem.

Our formal modeling advocates the separation of the *entropy extraction* phase from the *output generation* phase. We argue that the former is information-theoretic in nature, and should therefore rely on combinatorial and statistical tools and not on cryptography. On the other hand, we show that the latter phase can be implemented using any standard (non-robust) cryptographic PRG (which is turn can theoretically be built from any one-way function).

We also discuss the applicability of our architecture as a replacement for the current implementation of `/dev/(u)random` in Linux, and also examine using it as an architecture for pseudorandom generation on smartcards.

1 Introduction

Randomness is a very useful resource, and nowhere more than in cryptographic applications. Randomness is essential for secret keys, and inadequate source of randomness can compromise the strongest cryptographic protocol (e.g., see [GW96]). However, the reality is that procedures for obtaining random bits (called a *pseudo-random generators*) are often not designed as well as they could have been. This is unfortunate since a security flaw in this procedure translates into a security flaw in the whole system. Also, if this procedure does not have a rigorous security proof then there is no hope for such a proof for the whole system.

In this work we formalize the requirements that we believe are needed from a robust pseudo-random generator, and describe an architecture to realize these properties. Underlying our model is the view that randomness must be assessed *from the point of view of a potentially malicious observer*, and that this observer may have significant knowledge about inner workings of the system and the environment in which it operates, and it may even be able to influence that environment. Our goal is to ensure (to the extent possible) that even such observer cannot distinguish the output of the generator from an endless string of random bits. (Below we always refer to this observer as *the attacker*). Our formal model clarifies the conditions that must be met to achieve this goal, and our architecture demonstrates how to realize the security goal when the necessary conditions are met.

It is always assumed that the attacker knows the code of the generator itself. Hence, knowledge of the generators internal state would give the attacker the ability to predict its output. The heart of any design for robust pseudorandom generator is therefore to prevent an attacker from knowing the internal state (past, present, and future). This task is complicated by the fact that often times the attacker can get at the internal state by “external means”. (For example, by compromising the OS on which this generator runs and reading the memory content of the generator process.) Hence, many designs incorporate a “refresh” mechanism, by which the generator can modify its internal state using “new random data”. The intent is that the new state be completely unknown, even to an attacker with knowledge of the old state. Common pseudorandom generators therefore consist of two components:

- A function `next()` that generates the next output and then updates the state accordingly. The goal of this component is to ensure that if the attacker does not know the state, then it cannot distinguish the output (and new state) from random.
- A function `refresh(x)` that refreshes the current state using some additional input x . The goal of this component is to ensure that if the input has “high-entropy” (from the attacker’s point of view) then the resulting state is unknown to the attacker.

In some architectures these two components are mixed together, while in others they are separated. In this work we advocate separating these components, and argue that they are very different in nature. Some architectures includes also an “entropy estimation” component, that tries to estimate the entropy of the additional input before running the refresh function. (The intent is to run the refresh function only if the additional input is indeed “high-entropy”.) However, we believe that such component is counter-productive, since the quantity we are trying to estimate is the entropy *from the point of view of the attacker*. The attacker, however, is an adversarial entity on which we know next to nothing, so there is no conceivable way to actually estimate this entropy.¹ Therefore, at best the entropy estimator provides a false sense of security. At worst it can be actively manipulated by an attacker, causing the generator to refresh its state using data that is guessable by the attacker (or, in some cases, preventing refresh even though the available data is not guessable by the attacker). See [Section 5.2](#) for more discussion.

1.1 The model

The most important part of our formal model is modeling the capabilities of a malicious attacker. Somewhat informally, we consider an attacker that has access to the system where the generator is run, and has the following capabilities:

¹The designer of the generator may not even know much on the system in which this generator is to run, let alone the capabilities of an attacker on this system.

- Prompting the generator for output and observing this output. Namely, the attacker has an interface `next-bits()` that returns (say) the next m bits of output. (Throughout this paper, m is fixed to be some security parameter of the generator.)
- Observing and even influencing some of the data that is used to refresh the internal state of the generator. Roughly, we model this by allowing the attacker to specify the distribution from which the refresh data is drawn. This capability can be thought of as an interface `refresh(\mathcal{D})` that refreshes the internal state using an input that is drawn from the distribution \mathcal{D} (although our formal model is a bit different, see [Section 2](#)).
- Revealing and even modifying the internal state of the generator at will, i.e., an interface `set-state(s')` that changes the internal state to s' and returns the previous internal state.

The attacker interacts with the generator using these three interfaces, and the main security goal (i.e., *output randomness*) can be informally stated as follows:

If the attacker calls `refresh(\mathcal{D})` with a “high entropy distribution” \mathcal{D} , then the output of the generator from that point and until the next `set-state` call looks random to the attacker (even after it sees the result of that `set-state` call).

Note that this security goal in particular implies that once a “good” refresh occurred (i.e., a refresh with high entropy), the output of the generator will look random to the attacker as long as it does not obtain the internal state of the generator, *even if all later refreshes are “bad” in the sense that they are completely known or even controlled by the attacker.*

1.2 Additional requirements

In addition to making the output look random, it is also desirable that the generator protects the secrecy of the input that was used in the `refresh` operation. The reason is that the refresh data reveals things about the environment of the generator, and some of these things may be important to keep secret for various reasons. Consider for example a generator that uses the user’s key-strokes for refresh data, and imagine that the user types his/her password on the keyboard. Surely we don’t want an attacker to learn the user’s password by attacking the pseudorandom generator, so the generator should be built to protect this input. The level to which we can achieve input secrecy depends on the specifics of the attack. We assume that the attacker sees the output of the generator before and after the refresh operation, and in addition it may (or may not) know the internal state before and/or after the refresh. Then we have two cases:

- If the attacker does not know the internal state of the generator before the refresh operation, then we may be able to achieve *semantic security* for the refresh data. This means that the attacker cannot even check if a certain guess for the refresh data is correct or not.
- If the attacker knows the internal state of the generator before the refresh operation, it is easy to see that we cannot get semantic security, since an attacker can always test if a certain guess for the refresh data is consistent with the generator’s output after the refresh operation. In this case, the best that we can hope for is a “one wayness” property, where the attacker cannot recover the refresh data solely from its knowledge of the internal state before and after the refresh operation. (Assuming that the refresh data had enough entropy to foil an exhaustive search on all possible guesses.)

1.3 The construction

The construction uses two very different elements. One is a *randomness extraction function* $\text{extract}(x)$ that converts a “high entropy” input into a shorter “random” output. Namely, if the input is taken from a “high entropy” source, then the output is almost uniformly random. We comment that the function extract *need not be cryptographic at all*. (See [Section 2](#) and [Appendix A](#) for details.) The other element is a standard (non-robust) cryptographic PRG, that takes a short random seed and outputs a longer pseudorandom output, denoted $G(s)$. The output length of the extraction function equals the seed length of the cryptographic PRG (e.g., 160 bits if we use for cryptographic PRG, say, HMAC-SHA1 in counter mode).

Given these two elements, the construction is very simple: The internal state of the robust generator consists of a seed of the cryptographic PRG, which we denote by s , and let $m = |s|$ be the security parameter. In response to a call $\text{next}()$, the generator runs $G(s)$ to generate $2m$ pseudorandom bits, outputs the first m of them and replaces the state s by the last m bits. On a call $\text{refresh}(x)$, the generator xors $\text{extract}(x)$ into the current state, and then runs the cryptographic PRG once more, setting $s' \leftarrow G(s \oplus \text{extract}(x))$.

1.4 Relations to previous work

The problem of designing pseudorandom generators satisfying similar notions of robustness is well known and many designs have been suggested in the literature, some of them quite similar to ours. Thus the novelty of this paper is *not* in our particular design of a pseudorandom generator. Rather it is in our formal and rigorous *definition* for this generator and in our *proof* that our design satisfies this definition. In particular, our formal model incorporates the attacks that were discussed by Kelsey, Schneier, Wagner, and Hall in [\[KSWH98\]](#). We also mention that the current design can be thought of as the “stand-alone variant” of the proactive pseudo-random generators of Canetti and Herzberg [\[CH94\]](#). (In that work they considered having different nodes in a network sending each other (pseudo)random strings, and each node use the strings that it gets to refresh the state of its internal generator.) We mention some differences between our design and some previous works (such as the *Fortuna* pseudorandom generator of Ferguson and Schneier [\[FS03\]](#)) in the subsequent sections.

1.5 Organization

In [Section 2](#) below we discuss the extraction function and formalize the interface $\text{refresh}(\cdot)$ of the attacker that was sketched above. Then in [Section 3](#) we complete our formal modeling, providing a formal definition of a robust pseudorandom generator. In [Section 4](#) we describe our architecture and proves that it indeed realizes a robust pseudorandom generator. Finally, in [Section 5](#) we discuss several practical issues with the use of robust generators and suggest a few application areas where an architecture such as ours may be used. We mention some of the related work on randomness extraction on [Appendix A](#).

2 Using Entropy Extraction

In this section we model the attacker’s interfaces to the operation of refreshing the state. An important feature of our model is that it distinguishes between “normal” and “dysfunctional” conditions of the system. Roughly, this models the fact that there are times when the system can

harvest enough entropy to recover from an exposure of the state, and other times when it cannot (e.g., perhaps the attacker has installed a monitor on the system that tracks all the collected bits).

Following the common practice in cryptographic modeling, we let the attacker decide whether the system is in “normal” or “dysfunctional” state. Thus, the attacker in our model is given two interfaces, `good-refresh` that models refreshing the state under “normal” conditions, and `bad-refresh` that models refreshing the state under “dysfunctional” conditions. In both cases the attacker specifies the distribution from which the refresh data is drawn, but in the “dysfunctional” case the distribution can be arbitrary, while in the “normal” case we require that this be a “good distribution”.² The rest of this section is devoted to formalizing what we mean by a “good distribution”.

2.1 Good distributions for the refresh data

What we need from a “good distribution” \mathcal{D} is that drawing $x \leftarrow_{\mathcal{R}} \mathcal{D}$ and using x to refresh the state will result in future outputs of the generator looking random to the attacker, even if this attacker knows the current internal state of the generator. Attempting to formalize this requirement, let $F_s(x)$ be the function that returns the first output of the generator after refreshing the current state s with the refresh data x . Note that once the generator is fixed, the function F_s is a well defined deterministic function for all s .

Since we cannot say anything about the state of the generator before the refresh (for all we know it may be completely controlled by the attacker), we would like to require that the output of $F_s(x)$ looks random to the attacker *regardless of what s is*. Hence, for a distribution \mathcal{D} to be considered “good” (for some specific generator) we would like the distribution $F_s(\mathcal{D})$, which is induced by drawing $x \leftarrow_{\mathcal{R}} \mathcal{D}$ and computing $F_s(x)$, to be pseudorandom *for all s* . (We do not bother to make this notion completely formal, since we will not be using it in our model. It is only developed here to illustrate the problems with formalizing the notion of “good distributions”.)

In practice, in many cases people use a construction such as (say) $F_s(x) = \text{HMAC-SHA1}_s(x)$. Moreover, it seems that people make the assumption that *any distribution with enough entropy* is “good” for such constructions. We stress, however, that this assumption is in fact *false*. Indeed, it is easy to see that for *every* deterministic function $F_s(x)$ there exists a distribution \mathcal{D} (depending on F_s) with roughly $|x| - 1$ bits of entropy, such that $F_s(x)$ can be easily distinguished from the uniform distribution. (For example, let \mathcal{D} be the uniform distribution over all the strings x for which the first bit of $F_s(x)$ is zero.)

It follows that no single construction can “work” for all the high-entropy distributions. That is, for every design we have a “family of good distributions” for which that design works (that we denote \mathcal{H} for High-entropy), and \mathcal{H} is necessarily a proper subset of all the high entropy distribution. Clearly, we want \mathcal{H} to be as large as possible, so that hopefully it includes all the high-entropy distributions that arise in practice. However, recall that the designer of the generator typically knows very little about the environment in which it will be used (and even less about the capabilities of an attacker in this environment). To make the task of constructing, modeling and proving such generators more tractable, we therefore advocate breaking the operation of refreshing the state into two orthogonal components:

- One component is in charge of taking the refresh data x (which was hopefully drawn from one of these “good distributions”) and distilling from it a *truly random* nearly-uniform string d of length m (where m is the security parameter). Such procedures are commonly called *extractors* (c.f., [Sha02]), and hence we denote this procedure here by $d \leftarrow \text{extract}(x)$.

²To be completely formal we need to specify some standard encoding of distributions, to be used by the attacker with these interfaces. For simplicity, however, we ignore this unimportant issue here.

- The other component takes that (hopefully random) string d and uses it to modify the state.

These two components can be designed separately, and in principle they need not utilize the same tools. For the design of the first component we refer to the existing literature on entropy extraction, part of which is concerned with designing `extract` functions that work for “as general families \mathcal{H} as possible”. (For example, the model of Barak et al. [BST03] essentially lets the attacker specify the family \mathcal{H} so in some sense it is as general as you can get.) We briefly discuss in the appendix the relevant papers of Barak et al. [BST03] and Dodis et al. [DGH⁺04], and give some pointers to other works. In the rest of this work we simply assume that we have a good extraction function that works for a large enough family \mathcal{H} . Specifically, we use the following definition:

Definition 2.1 (Extraction functions). Let m be an integer, and let \mathcal{H} be a family of distributions over $\{0, 1\}^{\geq m}$ (i.e., the set of strings of length at least m). A function `extract` : $\{0, 1\}^{\geq m} \rightarrow \{0, 1\}^m$ is a \mathcal{H} -extractor if for every $\mathcal{D} \in \mathcal{H}$ and every $y \in \{0, 1\}^m$,

$$2^{-m}(1 - 2^{-m}) \leq \Pr_{x \leftarrow \mathcal{R}\mathcal{D}}[\text{extract}(x) = y] \leq 2^{-m}(1 + 2^{-m})$$

From now on we just fix some family \mathcal{H} and assume that we have a \mathcal{H} -extractor `extract`, and we show how to use it to build a robust pseudorandom generator using standard cryptographic tools.

2.2 Back to our model

As we explained above, our formal model is parameterized with a family \mathcal{H} of distributions that the system uses under “normal” conditions. Our security definition in Section 3 below and the construction later in Section 4 refer to the family \mathcal{H} and to some function `extract` that is a \mathcal{H} -extractor.

The attacker in our model is given the two interfaces `good-refresh` and `bad-refresh`. In principle, when using the interface `good-refresh` the attacker is required to specify a distribution $\mathcal{D} \in \mathcal{H}$, and when using `bad-refresh` it can specify an arbitrary distribution. We slightly simplify things by letting the attacker specify the actual input for the refresh algorithm in the “dysfunctional” case, instead of the distribution from which this input is taken. (This makes no difference as long as the distribution is efficiently sampleable.) Hence the attacker has the interfaces `good-refresh`(\mathcal{D}) that specifies a distribution $\mathcal{D} \in \mathcal{H}$ and `bad-refresh`(x) that specifies a single bit string x .

3 Robust Pseudorandom Generators

A robust pseudorandom generator consists of two functions:

$(r, s') \leftarrow \text{next}(s)$, where s is the current internal state. Returns (say) an m -bit string r , where m is the security parameter of the generator,³ and replaces the internal state by the new state s' .

$s' \leftarrow \text{refresh}(s, x)$, with x a string of length at least m and s the current internal state. Updates the internal state using the data x .

The security requirements from a robust generator are formulated via probabilistic games between two players: one player is *the system* that implements the generator, and the other is *the*

³We assume for simplicity that the generator returns every time exactly m bits, but both the definition and the construction generalize easily to the case of arbitrary-length output.

attacker that tries to attack the system. In the introduction we discussed several different properties that we expect from a robust pseudo-random generator. Instead of formulating each one via a separate formal definition, we choose to define security via the ideal-world/real-world paradigm (cf. [GMW91]). Namely, we have a real-world game that is meant to describe a real attacker that interacts with the robust generator, and we have an ideal-world game that is meant to capture “the most secure process that you could possibly get”. A construction is then deemed secure if no attacker can distinguish between interacting with the generator in the real world and interacting with the “secure process” in the ideal world. After presenting the formal definition, we briefly explain why it implies all the properties that we described in the introduction.

The real-world game. We model an attacker on the generator in the “real world” as an efficient procedure A that has four interfaces to the generator, namely `good-refresh(\cdot)`, `bad-refresh(\cdot)`, `set-state(\cdot)` and `next-bits(\cdot)`. Formally, the real-world game is parameterized by a security parameter m and a family of distribution \mathcal{H} (as described in Section 2 above). The game begins with the system player initializing the internal state of the generator to null, (i.e., $s \leftarrow 0^m$), and then the attacker interacts with the system using the following interfaces:

`good-refresh(\mathcal{D})` with \mathcal{D} a distribution in \mathcal{H} . The system draws $x \leftarrow_{\text{R}} \mathcal{D}$, sets $s' \leftarrow \text{refresh}(s, x)$, and updates the internal state to s' .

`bad-refresh(x)` with a bit string x . The system sets $s' \leftarrow \text{refresh}(s, x)$ and updates the internal state to s' .

`set-state(s')` with an m -bit string s' . The system returns to the attacker the current internal state s and then changes it to s' .

`next-bits(\cdot)`. The system runs $(r, s') \leftarrow \text{next}(s)$, replaces the internal state s by s' and returns to the attacker the m -bit string r .

The game continues in this fashion until the attacker decides to halt with some output in $\{0, 1\}$. For a particular construction $\text{PRG} = (\text{next}, \text{refresh})$, we let $\Pr[A(m, \mathcal{H})^{R(\text{PRG})} \rightarrow 1]$ denote the probability that A outputs the bit “1” after interacting as above with the system that implements the generator PRG and with parameters m, \mathcal{H} . (Here $R(\text{PRG})$ stands for the “real-world process” from above.)

The ideal-world game. The ideal-world game proceeds similarly to the real-world game, except that the calls that A makes to its interfaces are handled differently. Specifically, whenever A “expects to learn something new” from a `next-bits(\cdot)` or `set-state(\cdot)` call, the ideal process would return to A a new random m -bit string, independent of everything else.

The exception, of course, is that if A already knows the internal state due to a previous `set-state` call, then everything that it later sees until the next `good-refresh` call should be consistent with that internal state. This is done by having the system maintains a flag `compromised` that is set on a `set-state` call and reset on a `good-refresh` call, and the system behaves according to the prescribed robust generator when `compromised = true` and returns random strings when `compromised = false`.

Formally, the ideal-world game is parametrized by the same security parameter m and family of distribution \mathcal{H} as before. The game begins with the system player initializing $s \leftarrow 0^m$ and `compromised` \leftarrow `true` and then the attacker interacts with the system using the following interfaces:

`good-refresh(\mathcal{D})` with \mathcal{D} a distribution in \mathcal{H} . The system resets `compromised` \leftarrow `false`.

`bad-refresh(x)` with a bit string x . If `compromised = true` then the system sets $s' \leftarrow \text{refresh}(s, x)$ and updates the internal state to s' .

`set-state(s')` with an m -bit string s' . If `compromised = true` then the system returns to the attacker the current internal state s , and if `compromised = false` then it chooses a new random string $s \in_R \{0, 1\}^m$ and returns it to the attacker.

Either way, the system also sets `compromised` \leftarrow `true` and sets the new internal state to s' .

`next-bits()`. If `compromised = true` then the system runs $(r, s') \leftarrow \text{next}(s)$, replaces the internal state s by s' and returns to the attacker the m -bit string r . If `compromised = false` then the system chooses a new random string $r \in_R \{0, 1\}^m$ and returns it to the attacker.

The game continues in this fashion until the attacker decides to halt with some output in $\{0, 1\}$. For a particular construction $\text{PRG} = (\text{next}, \text{refresh})$, we let $\Pr[A(m, \mathcal{H})^{I(\text{PRG})} \rightarrow 1]$ denote the probability that A outputs the bit “1” after interacting as above with the system. (Here $I(\text{PRG})$ stands for the “ideal process” from above and note that we only use PRG in this game to answer queries that are made while the `compromised` flag is set to `true`.)

As we explained above, we say that a construction PRG is secure if no attacker can tell the difference between interacting with the system implementing PRG in the real world and interacting with the “ideal process”. Formally, we have the following:

Definition 3.1. We say that $\text{PRG} = (\text{next}, \text{refresh})$ is a *robust pseudorandom generator* (with respect to a family \mathcal{H} of distributions) if for every probabilistic polynomial-time attacker algorithm A , the difference

$$\left| \Pr[A(m, \mathcal{H})^{R(\text{PRG})} \rightarrow 1] - \Pr[A(m, \mathcal{H})^{I(\text{PRG})} \rightarrow 1] \right|$$

is negligible in the security parameter m .

3.1 Output randomness and input secrecy

We observe that the output-randomness requirement that we discussed in the introduction is hard-wired into the formal definition above. Indeed, the ideal process — when not compromised — always returns truly random strings. Thus, saying that an attacker cannot distinguish between the “real” and “ideal” processes in particular implies that it cannot distinguish the output of the real process from truly random bits.

It is not hard to see that [Definition 3.1](#) implies also the two input-secrecy requirements from the introduction. To see that, notice that as long as the ideal process is not compromised, it completely ignores the input to the refresh calls. Hence in the ideal world the attacker can neither recover the refresh data in a `good-refresh` call (regardless of whether the system is compromised before that call), nor can it check the refresh data in a `bad-refresh` call in which the system is not compromised. And since the attacker cannot distinguish the ideal process from the real one, it also cannot do these things when it interacts with the robust generator in the real world.

4 Our Construction

Now that we defined our formal model, we turn to presenting a simple construction that can be rigorously proven to satisfy our definition of a robust pseudorandom generator.

The main component of our construction — other than the extraction function — is a simple (non-robust) cryptographic PRG. A cryptographic PRG is a stateless function $G : \{0, 1\}^m \rightarrow$

$\{0, 1\}^{2m}$ such that $G(U_m)$ is computationally indistinguishable from U_{2m} (where m is a security parameter and U_i is the uniform distribution on $\{0, 1\}^i$). Such a generator can be built from symmetric ciphers and hash functions (e.g., either AES or HMAC-SHA1 in counter mode), the factoring problem [BM84], or even any arbitrary one way function [HILL99, GL89].

Below we write $(r, s') \leftarrow G(s)$ to denote that r is the first m bits in the output of $G(s)$ and s' is the last m bits. Also, we denote by G' the function that on input $s \in \{0, 1\}^m$ outputs only the first m bits of $G(s)$. Our pseudorandom generators operates as follows:

Operation of PRG. For security parameter m , given a \mathcal{H} -extractor $\text{extract} : \{0, 1\}^{\geq m} \rightarrow \{0, 1\}^m$ and a cryptographic non-robust PRG $G : \{0, 1\}^m \rightarrow \{0, 1\}^{2m}$, our robust PRG behaves as follows: It has a state $s \in \{0, 1\}^m$, and the functions `refresh` and `next` are defined as:

- `refresh(s, x)` returns $s' \leftarrow G'(s \oplus \text{extract}(x))$.
- `next(s)` returns $(r, s') \leftarrow G(s)$.

Our main theorem regarding this pseudorandom generator is the following:

Theorem 4.1. *Let m be a security parameter, let $\text{extract} : \{0, 1\}^{\geq m} \rightarrow \{0, 1\}^m$ and let \mathcal{H} be a family of distributions over bit strings, such that extract is a \mathcal{H} -extractor. Also, let G be a cryptographic PRG. Then the construction from above is a robust pseudorandom generator with respect to the family \mathcal{H} .*

Proof. Let us fix some efficient attacking algorithm A . We consider the following two experiments:

Expr. R A interacts with $R(\text{PRG})$ as in the real-world game from Section 3.

Expr. I A interacts with the ideal process $I(\text{PRG})$ as described in Section 3.

We want to prove that A outputs “1” in both experiments with almost the same probability. To this end, we consider another experiment, denoted **Expr. H** (for **H**ybrid). This is similar to **Expr. R**, except that it uses a truly random m -bit string to refresh the state in the good refresh calls (instead of the output of $\text{extract}(x)$ for $x \in_R \mathcal{D} \in \mathcal{H}$). Namely, in **Expr. H** the attacker A interacts with a modified process that works as follows:

`good-refresh(\mathcal{D})` with \mathcal{D} a distribution in \mathcal{H} . The system draws $d \in_R \{0, 1\}^m$, sets $s' \leftarrow G'(s \oplus d)$, and updates the internal state to s' .

`bad-refresh(x)` with a bit string x . The system sets $s' \leftarrow \text{refresh}(s \oplus \text{extract}(x))$ and updates the internal state to s' .

`set-state(s')` with an m -bit string s' . The system returns to the attacker the current internal state s and then changes it to s' .

`next-bits()`. The system runs $(r, s') \leftarrow \text{next}(s)$, replaces the internal state s by s' and returns to the attacker the m -bit string r .

On one hand, we show that the view of A in **Expr. H** is statistically close to its view in **Expr. R**. On the other hand, we show that to distinguish between **Expr. H** and **Expr. I** the attacker A would have to break the underlying cryptographic PRG.

Proving the first claim is fairly straightforward. Let q_r be a (polynomial) upper-bound on the number of `good-refresh(\cdot)` calls that A makes during the attack, and notice that the view of A is a deterministic function of the (q_r or less) m -bit strings that are the result of `extract(\cdot)` in these `good-refresh(\cdot)` calls. In **Expr. H** the distribution over these strings is the uniform distribution, whereas in **Expr. R** each of these string is set to `extract(x)` where x is chosen from (a convex combination of) the distribution in \mathcal{H} .⁴ But since the statistical distance between the uniform distribution and `extract(\mathcal{D})` is bounded by 2^{-n} for every $\mathcal{D} \in \mathcal{H}$, it follows that the statistical distance between the view of A in the two games **Expr. H** and **Expr. R** cannot exceed $q_r/2^n$.

To prove the second claim, we show that distinguishing **Expr. H** from **Expr. I** implies breaking the underlying cryptographic PRG. Let p_H be the probability that A outputs one in **Expr. H**, and similarly let p_I be the probability that A outputs one in **Expr. I**. We show a procedure B that breaks the cryptographic PRG G with advantage of at least $(p_H - p_I)/q$, where q is a polynomial bound on the total number of calls made by A to all of its interfaces. The procedure B gets as input two m -bit string r^*, s^* , and it tries to determine if they were chosen at random and independently from $\{0, 1\}^m$, or were set as $(r^*, s^*) \leftarrow G(s)$ for a random $s \in_R \{0, 1\}^m$.

The procedure $B(r^*, s^*)$ uses the attacker A as a subroutine, implementing for it the system. It begins by choosing at random an index $i^* \in_R \{1, 2, \dots, q\}$, and also setting $s \leftarrow 0^m$ and `compromised` \leftarrow `true`, and then it runs A , roughly answering the first $i^* - 1$ calls of A with random bit-strings, answering the i^* 'th call using the input (r^*, s^*) , and answering calls $i^* + 1$ and on as is done in **Expr. H**. More specifically, the i 'th call of A is answered as follows:

`good-refresh(\mathcal{D})` with \mathcal{D} a distribution in \mathcal{H} .

If $i < i^*$ then B chooses at random $s' \in_R \{0, 1\}^m$. If $i = i^*$ then B uses its input, setting $s' = s^*$. If $i > i^*$ then B chooses at random $d \in_R \{0, 1\}^m$ and sets $s' \leftarrow G'(s \oplus d)$ where s is the current internal state. Either way, B updates the internal state to s' and sets `compromised` \leftarrow `false`.

`bad-refresh(x)` with a bit string x .

If `compromised` = `true` or $i > i^*$ then B sets $s' \leftarrow G'(s \oplus \text{extract}(x))$. If `compromised` = `false` and $i < i^*$ then B chooses at random $s' \in_R \{0, 1\}^m$, and if `compromised` = `false` and $i = i^*$ then B uses its input, setting $s' = s^*$. Either way, B updates the internal state to s' (but does not modify the flag `compromised`).

`set-state(s')`.

B returns to the attacker the current internal state s and then changes it to s' , and also sets `compromised` \leftarrow `true`.

`next-bits()`.

If `compromised` = `true` or $i > i^*$ then B sets $(r, s') \leftarrow G(s)$. If `compromised` = `false` and $i < i^*$ then B chooses at random $r, s' \in_R \{0, 1\}^m$, and if `compromised` = `false` and $i = i^*$ then B uses its input, setting $r = r^*$ and $s' = s^*$. Either way, B replaces the internal state s by s' and returns to the attacker the m -bit string r .

The procedure B continues this way until A halts and outputs a bit, and then B outputs the same bit. To analyze the advantage of the procedure B , consider the $q + 1$ experiments $H^{(i)}$, $i = 0, 1, \dots, q$, where in experiment $H^{(i)}$ the first i calls of A to its interfaces are processed the way

⁴Note that the attacker A can specify different distributions from \mathcal{H} , depending on its internal randomness and its view thus far. Hence, each distribution in \mathcal{H} has some probability of being specified as the next distribution in a `good-refresh(\cdot)` call, so the overall distribution from which x is drawn is a convex sum of all the distributions in \mathcal{H} .

B processes queries for $i < i^*$, and the rest are processed the way B processes queries for $i > i^*$. Also, let $p^{(i)}$ be the probability that A outputs one in the experiment $H^{(i)}$.

It is clear that $p^{(q)} = p_h$, since B answers all the queries $i > i^*$ just as in **Expr. H**. We claim that also $p^{(0)} = p_0$. To see this, notice that the flag `compromised` is set by B in exactly the same way as in **Expr. I**, and the queries of A are always answered as in the “real world” with PRG when `compromised = true` (in both the run of B and in the experiment $H^{(0)}$). Also, all the queries $i < i^*$ with `compromised = false` are answered by B with random and independent bit strings, just as in the experiment $H^{(0)}$. (The only tricky case is a `set-state(·)` query, but notice that B returns the “current state s ”, which is a random bit string that was never used by B in any other query, since we have $i < i^*$ and `compromised = true`.)

It is also clear that when B chooses some i^* and its input is chosen at random ($r^*, s^* \in_R \{0, 1\}^m$) then B outputs one with probability exactly $p^{(i^*)}$. Moreover, it is not hard to see that when B chooses some i^* and its input is chosen as the output of G ($(r^*, s^*) \leftarrow G(s)$ for $s \in_R \{0, 1\}^m$) then B outputs one with probability $p^{(i^*-1)}$. Specifically, for the latter case we note the following:

- If the i^* call of A is `good-refresh(·)` then B would set the internal state to $s' \leftarrow G'(s)$ where s is the randomness that was used to generate the input for B , whereas in the experiment $H^{(i^*-1)}$ the new state would be set to $s' \leftarrow G'(s \oplus d)$ with s the prior state and a newly random choice $d \in_R \{0, 1\}^m$, so the distribution of s' is the same in both.
- If the i^* call of A is `bad-refresh(x)` with `compromised = false`, then again B would set the internal state to $s' \leftarrow G'(s)$ where s is the randomness that was used to generate the input for B , whereas in the experiment $H^{(i^*-1)}$ the new state would be set to $s' \leftarrow G'(s \oplus \text{extract}(x))$ with s the previous state. However, since `compromised = false` and since all the calls upto $i^* - 1$ with `compromised = false` were processed with random strings, it follows that in this case the previous state is itself uniform in $\{0, 1\}^m$ (and independent of everything else), so again the distribution of s' is the same in both.
- If the i^* call of A is `next-bits()` with `compromised = false`, then B would set $(r, s') \leftarrow G'(s)$ where s is the randomness that was used to generate the input for B . In the the experiment $H^{(i^*-1)}$ these values are would be set to $(r, s') \leftarrow G'(s)$ with s the previous state. Again, since `compromised = false` and since all the calls upto $i^* - 1$ with `compromised = false` were processed with random strings, the previous state is itself uniform in $\{0, 1\}^m$ (and independent of everything else), so the distribution of (r, s') is the same in both.

From all the above it follows that the advantage of B is

$$\Pr_{s, \text{coins}} [B(G(s)) \rightarrow 1] - \Pr_{r^*, s^*, \text{coins}} [B(r^*, s^*) \rightarrow 1] = \sum_{i^*=1}^q \frac{p^{(i^*-1)} - p^{(i^*)}}{q} = \frac{p_0 - p_h}{q}$$

□

5 Practical Considerations

5.1 Drawing and extracting from random sources

In our formal model we have the system draws from a distribution \mathcal{D} only when it is queried with `good-refresh(\mathcal{D})`. Most realistic implementations, however, would likely draw from their entropy sources frequently (e.g., every few milliseconds), but will only modify the internal state with use the

data so gathered every so often (e.g., once every few minutes, or even once an hour). Conceptually, we would like to think of this process as buffering all the data until a refresh is performed, and then using it all at once. However, it is not realistic to expect that so much data will be buffered between refresh operations. Instead, implementations are likely to use an extraction function that can process the data in an on-line fashion, never keeping more than a few dozens (or a few hundreds) bytes of state.

5.2 To refresh or not to refresh

An important issue in the deployment of robust pseudo-random generator is to decide when to run the refresh algorithm. On the one hand, refreshing very often pose the risk of using refresh data that does not have enough entropy, thus allowing an attacker that knows the previous internal state to learn the new state by exhaustively searching through the limited space of possibilities (cf. the “State Compromise Extension Attacks” from [KSWH98]). On the other hand, refreshing the state very infrequently means that it takes longer to recover from a compromise.

We stress that refreshing the state only helps if an attacker was able to compromise the previous internal state but not the new state.⁵ For example, when the system running the generator was infected with a virus that leaked the previous internal state, but later the virus was removed so the new state can no longer leak. Indeed, we believe that in most all real-life systems, the frequency of events in which an attacker broke into the system and then “left it” is very low. Moreover, if the “system cleanup” requires explicit human interaction, then the same human interaction can possibly be used also to generate sufficient entropy before refreshing the state (if nothing else, by having the human randomly hitting the keyboard for a little while). It seems therefore that in most realistic settings, the default frequency for refreshing the state could be very low (e.g., once an hour).

As we discussed in the introduction, we believe that implementing an automated entropy estimation routine for the purpose of scheduling a state refresh is counter productive, since measuring of entropy *from the point of view of an attacker* is well beyond what can be expected from a computer program. Instead, we advocate having a very low default refresh frequency (and letting an administrator overwrite the default at their own risk).

The Fortuna heuristic. The Fortuna architecture by Ferguson and Schneier [FS03] uses a sophisticated heuristic to automatically schedule refreshes without having to estimate the entropy. Roughly, the refresh data is divided between several “pools”, and the different pools are used with different frequencies. In particular, they use 32 pools where the i^{th} pool is used every 2^i refreshes. Intuitively, this heuristic is supposed to “eat the cake and have it too” in the sense that it should enjoy both the recovery speed of a frequent refreshes and the security advantages of infrequent refreshes.

However, we point out that there is a strong assumption underlying this heuristic, and the heuristic may fail to enjoy neither recovery speed nor security if this assumption fails. Specifically, the heuristic assumes that the different pools are “reasonably independent”. Making such independence assumption seems to be assuming quite a lot, especially if we recall that we must consider the distribution of the different entropy pools from the attacker’s point of view. This is even more so due to the specific way that the multi-pool idea is implemented in Fortuna. Namely, each source spreads its bytes among all of the pools in a cyclical / “round robin” fashion (see [FS03, Sec 10.5.2,

⁵ This is because for an attacker that did not know this state, the generator will remain secure even if it is always refreshed with zero entropy (or even with adversarially chosen data).

Page 169]), so in many cases “the next bytes” in pool $i + 1$ will be highly correlated to “the next bytes” in pool i .

To see what goes wrong when the independence assumption fails, consider an extreme case in which there is just a single source, and that source has just one bit of entropy every 32^{nd} time that it is called and zero entropy otherwise. Namely, from the point of view of an attacker that knows all the outputs of the source thus far there are only two possibilities for its output in the next call, and the output of the source in the 31 calls after this time that are completely determined by this next output (say for simplicity that the next 32 calls all return exactly the same sequence of bytes). Now, if the system refreshes its state using the Fortuna heuristic, then it is not hard to see that the attacker can mount a “state-compromise extension attack” (a-la-[KSWH98]), and will only need to try two possible values for the refresh data. This means that regardless of how long the system runs, it will never recover to a secure state, even though it could have recovered to a secure state if it only uses the least frequent pool (i.e., the most conservative estimate) to refresh instead of using the Fortuna heuristic.

Although this is a very extreme example, we believe that in general it is better to refresh the internal state very rarely when possible, as opposed to trying to speed things up at the expense of making extra assumptions on the distributions of the various sources. Because refreshes do not need to occur too often, in the vast majority of cases even refreshing once every hour will still be good enough. (We also comment that if the independence assumption does hold, one may be able to use this in more ways than the use in the Fortuna heuristic. This is because extracting randomness from several *independent* pools of entropy is an easier problem than extracting randomness from a single entropy pool or from related entropy pools, see [BIW04].) **Note:** Our model assumes that every good refresh is made from a distribution in \mathcal{H} regardless of the outcome of the previous good refresh. Hence we do assume a conditional entropy requirement between all the good refreshes, which is a slightly weaker than the “reasonable independence” requirement. However, we note that it is much more reasonable to assume independence or conditional entropy between data that is collected in disjoint (and far apart) time intervals than between data that is collected in roughly the same time (as in the case of Fortuna).

Refreshing after reboot. One case where the “once-a-hour rule of thumb” cannot be applied is the initial refresh of the system when it is first started (or when it is rebooted). Clearly, in most cases we cannot stop the boot process for an hour to get entropy. Even in this case we argue that run-time entropy estimation is a bad idea (for the same reasons that it is a bad idea in other cases). Instead, several system-engineering solutions should be applied, (e.g., trying to save the random state from previous boot, or from hardware resources that are only available at boot time). As a last resort, we suggest setting a system parameter that specifies the time until first refresh, with a reasonable default (e.g., 10 seconds?) that an administrator can override.

Another idea is to use here “exponentially increasing intervals”, similarly to the Fortuna heuristic from above. Namely, at boot time one could schedule the first refresh after ten seconds, then after twenty more seconds, then forty, etc., until the system reaches the steady state in which it only refreshes the state once an hour.⁶ Assuming that the sources have more or less constant rate and that data drawn from the sources in non-overlapping time intervals is “reasonably independent”, this lets the system reach a secure state after no more than twice what is strictly needed. Of course when using this approach one needs to ensure that the “reasonable independence” condition can be assumed. One possible heuristic may be that after sampling data for an interval of x seconds,

⁶Of course we would keep sampling from the source at regular intervals (e.g., every few milliseconds) but we will accumulate more and more data from one refresh to the next.

the system should wait another x seconds without sampling any data, to increase the likelihood that the data sampled next would be independent from the previous sample. This may be a good heuristic to apply in other cases as well.

5.3 Discussion and relevance to `/dev/random`

The current implementation of Linux has two sources of randomness: `/dev/random` which is supposed to provide information-theoretic (i.e. statistical) security, and `/dev/urandom` which is supposed to provide computational security. Specifically, the manual reads as follows:

```
When read, the /dev/random device will only return random
bytes within the estimated number of bits of noise in the
entropy pool. /dev/random should be suitable for uses
that need very high quality randomness such as one-time
pad or key generation. When the entropy pool is empty,
reads to /dev/random will block until additional environ-
mental noise is gathered.
```

```
When read, /dev/urandom device will return as many bytes
as are requested. As a result, if there is not sufficient
entropy in the entropy pool, the returned values are theo-
retically vulnerable to a cryptographic attack on the
algorithms used by the driver. Knowledge of how to do
this is not available in the current non-classified liter-
ature, but it is theoretically possible that such an
attack may exist. If this is a concern in your applica-
tion, use /dev/random instead.
```

Given this documentation, it is expected that programmers will often use `/dev/random` in a security-related applications. We strongly disagree with that approach, and believe that in most situations, security will be enhanced if both `/dev/random` and `/dev/urandom` perform the same function, which is to run an instance of the general scheme described in this paper. Our reasons are the following:

1. It is not at all clear that `/dev/random`, whose implementation relies on an entropy estimator and the cryptographic hash function MD5 indeed provides information-theoretic security. Indeed, we suspect that it sometimes doesn't.
2. A design like ours allows for much more conservative entropy estimates (one needs fresh randomness much less quickly) than the current design of `/dev/random` and thus we believe will result in better chances of recovering from a leakage of the internal state.
3. The blocking behavior of `/dev/random` introduces uncertainty in the scheduling of security-related programs. This uncertainty may be exploited for attacks by an adversary.

The manual page from above seem to suggest that security of outputs from `/dev/urandom` relies on unproven assumption, whereas the security of `/dev/random` does not. In our view, there is nothing further from the truth. Leaving aside the fact that `/dev/random` uses MD5 as an entropy extractor, the “information theoretical security” that it provides relies on a heuristic approach to

entropy estimation. As we said several times in this note, the goal of that heuristic is to estimate the entropy from the attacker’s point of view, and it operates without any clue as to the environment in which it runs and the capabilities of the attacker. It all but certain that there are environments in which this estimation is far too optimistic. We believe that is far safer to trust the security of a published cryptographic design such as HMAC-MD5 or CBC-AES than to trust the entropy estimation heuristic. We also remark that the design in this paper can be instantiated with different primitives, and so can be made to rely on security of different problems (e.g., symmetric or public-key cryptography) or different key sizes. This can allow to use different tradeoffs between security and efficiency in different systems.⁷

Moreover, we point out that it may be the case that `/dev/random` does *not* provide information-theoretic security *even if the entropy estimator is correct*. This is because both streams use entropy from the same sources: even when `/dev/random` blocks, `/dev/urandom` continues to run and output bits. Eventually, the output of `/dev/urandom` may completely determine the contents of the entropy pools that both streams use (in an information-theoretic sense). To illustrate that point, consider running two applications on the same machine: Application A that uses `/dev/random` for a one-time-pad and Application B that keeps requesting bits from `/dev/urandom` and broadcasting them to everyone. From an information-theory perspective, it may very well be that the broadcast bits will eventually uniquely determine the context of the entropy pool, including the one-time-pad that was previously used by Application A.⁸

5.4 Other use cases

We note that there are other scenarios besides `/dev/random` where our architecture may be of use. One example is a smartcard that has no entropy generator of its own and can get what is supposed to be fresh randomness whenever it is connected to a reader. The properties of our architecture ensure that as long as this card gets randomness from an honest reader every once in a while, there is no harm in letting it act also on “randomness” from a malicious readers. This may be sometimes safer than including an entropy-generator in a smart-card, since those are often prone to attacks or malfunction.

Acknowledgments

We thank David Wagner for critically commenting on an earlier version of this paper. We also thank the participants of a long thread on pseudorandom generation in the `sci.crypt` usenet newsgroup in the fall of 2004, for motivating us to write this paper.

References

- [BIW04] Boaz Barak, Russell Impagliazzo, and Avi Wigderson. Extracting randomness from few independent sources, 2004. To appear in FOCS’ 04.

⁷Fortunately, pseudorandom generators are only supposed to output random-looking data and hence there is no compatibility issue in allowing different machines to use different implementations of `/dev/{u}random`.

⁸Note that this problem still remains even if the two streams use syntactically different pools, as long as steps are not taken to ensure that the pool of `/dev/random` has sufficient entropy even conditioned on the contents of the pool of `/dev/urandom`. If one desires information-theoretic security then we should estimate the amount of this conditional entropy when applying the extractor (which further demonstrates the problems with entropy estimation).

- [BM84] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, November 1984. Preliminary version in FOCS ’82.
- [BST03] Boaz Barak, Ronen Shaltiel, and Eran Tromer. True random number generators secure in a changing environment. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 166–180, 2003. LNCS no. 2779.
- [CH94] R. Canetti and A. Herzberg. Maintaining security in the presence of transient faults. In *Crypto ’94*, volume 839, pages 425–438, 1994. LNCS No. 839.
- [DGH⁺04] Yevgeniy Dodis, Rosario Gennaro, Johan Hastad, Hugo Krawczyk, and Tal Rabin. Randomness extraction and key derivation using the cbc, cascade and hmac modes. 2004.
- [FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley, New York, NY, USA, 2003.
- [GL89] Oded Goldreich and Leonid A. Levin. A hard-core predicate for all one-way functions. In *Proc. 21st STOC*, pages 25–32. ACM, 1989.
- [GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, July 1991. Preliminary version in FOCS’ 86.
- [GW96] Ian Goldberg and David Wagner. Randomness and the netscape browser. *Dr. Dobb’s Journal*, pages 66–70, 1996.
- [HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999. Preliminary versions appeared in STOC’ 89 and STOC’ 90.
- [KSWH98] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudorandom number generators. *Lecture Notes in Computer Science*, 1372:168–188, 1998.
- [Sha02] Ronen Shaltiel. Recent developments in extractors. *Bulletin of the European Association for Theoretical Computer Science*, 2002. Available from <http://www.wisodm.weizmann.ac.il/~ronens>.
- [TV00] L. Trevisan and S. Vadhan. Extracting randomness from samplable distributions. In *Proc. 41st FOCS*, pages 32–42. IEEE, 2000.

A More on Randomness Extraction

We now return to the issue of extracting “truly random” strings from high-entropy sources. Recall that in our context, we need a function $\text{extract} : \{0, 1\}^{\geq m} \rightarrow \{0, 1\}^m$ that returns an almost uniformly distributed string under “normal conditions” (and we do not care what it does under “dysfunctional conditions”). These “normal conditions” are codified by a family \mathcal{H} of distributions on bit strings, such that we can assume that the refresh data is drawn from some distribution $\mathcal{D} \in \mathcal{H}$ under “normal conditions”.

The design of the function `extract` depends on the family of distributions \mathcal{H} that one wishes to extract from, which depend of course on the type of data that we will use to refresh the generator. Below we briefly describe the work of Barak et al. [BST03] that sets up a model obtaining essentially the most general family \mathcal{H} and shows a rather efficient combinatorial construction that can be proven to work in that model. Later we discuss some other approaches to entropy extraction, and in particular describes briefly the work of Dodis et al. [DGH⁺04] that deals with using constructions such as CBC-AES and HMAC-SHA1 for extraction.

We remark that in addition to these works, there is an extensive literature on the problem of randomness extraction for the purposes of executing probabilistic algorithms (e.g., see the survey [Sha02] and the references therein).

A.1 The Barak-Shaltiel-Tromer model

The work [BST03] deals with extracting truly random bits from sources that have high entropy but are far from uniform. The goal is to construct a \mathcal{H} -extractor (cf. Definition 2.1) for a family of distributions that captures all the high-entropy distributions that we likely to see in practice.

Before proceeding further we mention the minor technical problem that Shannon entropy is not a good measure for the quality of distribution in our case. For example, a distribution that outputs the n -bit all-zero string with probability $1/2$, and otherwise outputs a uniform n -bit string, has about $n - 1$ bits of entropy, but it is easy to see that it is not possible to extract from it more than one bit of true randomness. It turns out that the right measure to use is the *min-entropy* of a source, which is defined as $\log(1/p)$ where p is the probability of the most likely output of that source. Hence, we require that all the distributions in \mathcal{H} will have more than m bits of min-entropy.⁹

Recall further that the designer of the extraction function typically knows very little about the environment in which it will be used. Therefore, the model of Barak et al. *lets the attacker specify the family \mathcal{H}* , subject to two constraints: (a) all the distributions in \mathcal{H} must have (significantly) more than m bits of min-entropy¹⁰, and (b) the family \mathcal{H} is not too large: at most 2^t distributions where t is a parameter. The parameter t can be thought of as the degree of influence that the attacker has on the environment under “normal conditions”. Namely, if there are t aspects of the environment that the attacker can control (and each of these is a boolean flag) then the attacker can choose one of 2^t different distributions depending on how it sets these flags.

In more details, the work [BST03] utilizes *randomized* extraction functions, $\text{extract} : \text{Coins} \times \{0, 1\}^n \rightarrow \{0, 1\}^m$, (for some $n > m$) and considers the following “game” between the designer and the attacker (with parameter t):

1. The attacker chooses a family of 2^t distributions, $\mathcal{H} = \{\mathcal{D}_1, \dots, \mathcal{D}_{2^t}\}$.
2. The designer chooses “once and for all” the randomness $s \leftarrow_{\text{R}} \text{Coins}$ for the extraction function. The coins s are made public (and in particular given to the attacker).

The randomized construction `extract` is considered good if for every family \mathcal{H} of 2^t distributions, for all but a 2^{-m} fraction of the coins $s \in \text{Coins}$, the deterministic function $\text{extract}_s(\cdot) = \text{extract}(s, \cdot)$ is a \mathcal{H} -extractor as per Definition 2.1. Namely, no matter how the attacker chooses its family \mathcal{H} in Step 1, the resulting extraction function extract_s extracts nearly uniform bits from every distribution in \mathcal{H} (except perhaps with insignificant probability over the choice of the coins in Step 2). It was

⁹One can also use the Renyi entropy of order two instead of the min-entropy, and these two quantities are closely related.

¹⁰Recall that \mathcal{H} describe the distributions over refresh data *under normal working conditions*, so it is reasonable to require that these distributions have high entropy.

shown in [BST03] how to construct practical randomized extraction functions that work as long as every distribution in \mathcal{H} has more than $2t + 4m$ bits of min-entropy. This construction is not cryptographic, and in particular it does not rely on any hardness assumption.

Some additional comments. We briefly mention that the result of Barak et al. from [BST03] is slightly stronger than what is implied by the text above. In particular, it can be shown that any extractor that works for a family \mathcal{H} also works for every distribution in the convex hull of \mathcal{H} , so one does not really have to think of the aspects that the attacker controls as being just boolean flags. Also, for our purposes we do not necessarily have to think of the different samples from the sources as independent, as long as we can assume that the distribution of the outputs *conditioned on the previous samples* belong to the family \mathcal{H} . Finally, do not claim that the model in [BST03] is necessarily “the right model” to use in this context. Indeed, we expect that more work has to be done before we have a model for randomness extraction that is both theoretically sound and “practically interesting”.

A.2 Other approaches to entropy extraction

Extraction using several independent sources. If one assume that we are given access to *several* sources of high entropy that are *independent* then the impossibility result for a single deterministic extraction function can be avoided, and it is possible to use a single deterministic function $\text{extract}(\cdot)$ to extract randomness from such sources. See [BIW04] and the references therein for discussion of this model.

Using cryptographic modes of operation. Many systems today use for entropy extraction cryptographic functions in some mode of operation. Although it is not at all clear what properties are needed from the cryptographic functions themselves for this to work, one can at least examine the properties of the mode of operation, assuming that the cryptographic primitive is replaced by a truly random function. Indeed, this was recently studied by Dodis, Gennaro, Håstad, Krawczyk and Rabin in [DGH⁺04]. Specifically, they studied the CBC- Π construction with Π a truly random permutation over $\{0, 1\}^m$, and the HMAC- f construction where f is a truly random “compression function” from $\{0, 1\}^n$ to $\{0, 1\}^m$, and analyzed the extraction properties of these functions.

Their results indicates that CBC- Π for a random permutation Π is a “reasonably good” extractor. Namely, for every distribution \mathcal{D} on $\{0, 1\}^{mL}$ (for some $L > 1$) that has more than $2m$ bits of min-entropy, and with very high probability over the choice of Π , the statistical distance between CBC- $\Pi(\mathcal{D})$ and U_m is bounded by roughly $L/2^{m/2}$. (This means more or less that for any m -bit string y , $\Pr_{x \leftarrow \mathcal{D}}[\text{CBC-}\Pi(x) = y] = 2^{-m}(1 \pm \frac{L}{2^{m/2}})$). Their results for HMAC- f are somewhat weaker (but similar in spirit).

Extraction from sampleable sources. Trevisan and Vadhan [TV00] put forward a theoretically very interesting model of randomness extraction from *sampleable sources*. The idea is that distributions arising in nature must be efficiently sampleable. They give both a construction of a function ensemble (as in the later work [BST03]) and a construction of a single function based on complexity assumptions (which unfortunately has rather weak parameters). The main problem in this approach is that complexity of the extraction function must be greater than the complexity of the sampling algorithm.