# A model and Architecture for Pseudo-Random Generation and Applications to `/dev/random`

Boaz Barak[*]
Department of Computer Science
Princeton University
boaz@cs.princeton.edu

Shai Halevi
IBM, T.J. Watson Research Center
shaih@alum.mit.edu

## 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/control of the generator's entropy source. Our model and architecture have the following properties:

- *Resilience.* The generator's output looks random to an observer with no knowledge 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, even if the observer learns the internal state at a later time.
- *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 could therefore rely on combinatorial and statistical tools rather than on cryptography. On the other hand, we show that the latter can be implemented using any standard (non-robust) cryptographic PRG.

We also discuss the applicability of our architecture for applications such as `/dev/(u)random` in Linux and pseudorandom generation on smartcards.

[*]Work was done while in the Institute for Advanced Study and partially supported by NSF grants DMS-0111298 and CCR-0324906.

**Categories and Subject Descriptors:** D.4.6 [Software]: Security and Protection

**General Terms:** Security

**Keywords:** `/dev/random`, Entropy, Mixing functions, Pseudorandomness, Smart-cards, True randomness.

## 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 [8]). However, the reality is that procedures for obtaining random bits (called *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 pseudorandom 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 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. 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.

Below we refer to this potentially malicious observer as *the attacker*. It is always assumed that the attacker knows the code of the generator itself. Hence, knowledge of the generator's internal state would give the attacker the ability to predict its output. The heart of any design for robust pseudorandom generator is therefore devoted to preventing an attacker from learning 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 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 separate. In this work we advocate separating these components, and argue that they are very different in nature. Some architectures include also an "entropy estimation" component, that tries to estimate the entropy of the additional input before running the refresh function, with the intent of running the refresh function only if the additional input is indeed "high-entropy". 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 an attacker. Loosely speaking, we consider an attacker that has access to the system where the generator is run, and has the following capabilities:

- 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 in 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 returns to the attacker the internal state and then changes this state to $s'$.

[1]The 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.

The attacker interacts with the generator using these three interfaces, and the main security goal 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 must look random to the attacker (even given the internal state obtained from 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 *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: Even knowing the internal state of the generator before and after the refresh operation the attacker should not be able to recover a "high entropy" refresh data,[2] and not knowing the state before the refresh the attacker should not even be able check if a certain guess for the refresh data is correct or not.

## 1.3 The construction

The construction uses two distinct elements. One is a *randomness extraction function* 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 random seed and outputs a longer (e.g., of size twice the size of the seed) pseudorandom output, denoted $\mathsf{G}(s)$. The output length of the extraction function equals the seed length of the cryptographic PRG (e.g., 128 bits if we use for cryptographic PRG, say, AES 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 next(), the generator runs $\mathsf{G}(s)$ to generate $2m$ pseudorandom bits, outputs the first $m$ of them and replaces the state $s$ by the last $m$ bits. On a call refresh$(x)$, the generator xors extract$(x)$ into the current state, and then runs the cryptographic PRG once more, setting $s' \leftarrow \mathsf{G}(s \oplus \mathsf{extract}(x))$.

## 1.4 Relation to previous work

The problem of designing pseudorandom generators satisfying similar notions of robustness is well known and many

[2]We can actually achieve the property of "entropic security", see [6].

designs have been suggested in the literature, some quite similar to ours. In particular, the *Fortuna* architecture of Ferguson and Schneier [7] is very similar to what we propose in this paper. (Some minor differences are discussed in Section 5.2.) 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 [13].

In general, robust pseudorandom generation can be viewed as an extension of the notion of *forward security*, adding the feature of automatic (or proactive) recovery from compromise. Indeed, the construction that we describe in Section 4 is very similar to the forward-secure generator of Bellare and Yee [2]. 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 [4]. (In that work they considered robust pseudorandom generation is a distributed network where nodes can help each other recover from state compromise.)

Finally, this work is concerned mostly with the *algorithmic* aspects of robust pseudorandom generation. For an excellent exposition of the *system* aspects we refer the reader to Gutmann's paper [11]. (However, we disagree with some of Gutmann's opinions regarding the algorithmic aspects. See discussion in Section 5.2.)

### 1.5 Organization

In Section 2 below we formalize the interface $\mathsf{refresh}(\cdot)$ 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. STATE REFRESH AND ENTROPY EXTRACTION

In this section we formally model the attacker's interfaces to the state-refresh operation. 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., if the attacker installs a monitor on the system to track 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, $\mathsf{good\text{-}refresh}$ that models refreshing the state under "normal" conditions, and $\mathsf{bad\text{-}refresh}$ 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". This section is mostly about formalizing what we mean by "good distributions".

### 2.1 Good distributions for the refresh data

What we need from a "good distribution" $\mathcal{D}$ is that drawing $x \leftarrow_{\mathrm{R}} \mathcal{D}$ 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. To illustrate the difficulties in formalizing this notion, 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.* Namely, to define what it means for a distribution $\mathcal{D}$ to be "good" we could consider the following game:

1. The attacker picks the current state $s$,
2. A refresh data $x$ is drawn from $\mathcal{D}$,
3. The attacker is given $F_s(x)$.

We would like to say that a necessary condition for $\mathcal{D}$ to be "good" is that the attacker cannot distinguish the value $F_s(x)$ in this game from a truly random string. 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 an (efficiently sampleable) distribution $\mathcal{D}_s$ (depending on $F_s$) with roughly $|x| - 1$ bits of entropy, such that $F_s(x \leftarrow_{\mathrm{R}} \mathcal{D})$ can be easily distinguished from the uniform distribution. (For example, let $\mathcal{D}_s$ 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 (in the sense of the game from above). Of course, we only care about the construction "working" for distributions that are likely to arise in practice, but it is not at all clear how to formally define such distributions. Moreover, recall that we are interested in the distribution of refresh data from the attacker's perspective, on which we know very little.

To make the task of constructing, modeling and proving such generators more tractable, we therefore propose the following approach: We introduce a parameter $\mathcal{H}$ which is supposed to capture a family of "good distributions", and formally define what it means for a generator be secure with respect to a given family $\mathcal{H}$ (cf. Definition 3.1 in Section 3).[3] We then advocate splitting the construction of the generator into two orthogonal components:

- One component is a procedure for processing the refresh data $x$, trying to distill from it a nearly-uniform string $d$ of length $m$ (where $m$ is the security parameter). Such procedures are commonly called *extractors* (cf., [14] and references therein). In Definition 2.1 we formalize what it means for a procedure $\mathsf{extract}(x)$ to "work" for a given distribution family $\mathcal{H}$.

[3]The notation $\mathcal{H}$ is meant to suggest High-entropy.

- The other component takes the (hopefully random) string $d$ and uses it in the operation of the generator. We give one simple example of how it can be constructed (using non-robust cryptographic PRG) in Section 4.

The randomness extraction component of a generator needs to distill (something close to) true randomness from the family $\mathcal{H}$ of distributions that are likely to arise in the system under "normal working conditions". Its design and analysis must therefore take into account at least some of the specifics of the target system(s). For the design of this component we defer to previous works such as [1, 5] (which work for essentially as general families $\mathcal{H}$ as possible) and the extensive research on randomness extractors as surveyed in [14].

We note that extractors can be built using combinatorial and statistical tools, and in principle *need not use any cryptography at all* (although there may be efficiency gains in using cryptographic components for this task). We describe some of these previous works in the appendix, and we also give some hints on concrete implementations in Section 4. Formally, we make the following requirements from a randomness extraction function:

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* $\mathsf{extract} : \{0,1\}^{\geq m} \to \{0,1\}^m$ *is an $\mathcal{H}$-*extractor *if for every $\mathcal{D} \in \mathcal{H}$ and every $y \in \{0,1\}^m$,*

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

## 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. 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. (These two formulations are equivalent as long as the distributions are 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:

$(r, s') \leftarrow \mathsf{next}(s)$, where $s$ is the current internal state. Returns (say) an $m$-bit string $r$, where $m$ is the security parameter of the generator,[4] and replaces the internal state by the new state $s'$.

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

[4]We assume for simplicity that the generator always output exactly $m$ bits, but both the definition and the construction generalize easily to the case of variable-length output.

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 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. [10]). 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(). 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 $x \leftarrow_R D$, sets $s' \leftarrow \mathsf{refresh}(s, x)$, and updates the internal state to $s'$.

bad-refresh($x$) with a bit string $x$. The system sets $s' \leftarrow \mathsf{refresh}(s, x)$ 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(). The system runs $(r, s') \leftarrow \mathsf{next}(s)$, 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 $\mathsf{PRG} = (\mathsf{next}, \mathsf{refresh})$, we let $\Pr[A(m, \mathcal{H})^{R(\mathsf{PRG})} \to 1]$ 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 $R(\mathsf{PRG})$ 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() 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 $\mathsf{compromised} = \mathsf{true}$ and returns random strings when $\mathsf{compromised} = \mathsf{false}$.

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 $\mathsf{compromised} \leftarrow \mathsf{true}$ and then the attacker interacts with the system using the following interfaces:

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

$\mathsf{bad\text{-}refresh}(x)$ with a bit string $x$. If $\mathsf{compromised} = \mathsf{true}$ then the system sets $s' \leftarrow \mathsf{refresh}(s, x)$ and updates the internal state to $s'$. Otherwise (if $\mathsf{compromised} = \mathsf{false}$) it does nothing.

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

Either way, the system also sets $\mathsf{compromised} \leftarrow \mathsf{true}$ and sets the new internal state to $s'$.

$\mathsf{next\text{-}bits}()$. If $\mathsf{compromised} = \mathsf{true}$ then the system runs $(r, s') \leftarrow \mathsf{next}(s)$, replaces the internal state $s$ by $s'$ and returns to the attacker the $m$-bit string $r$. If $\mathsf{compromised} = \mathsf{false}$ then the system chooses a new random string $r \leftarrow_{\mathrm{R}} \{0,1\}^m$ 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 $\mathsf{PRG} = (\mathsf{next}, \mathsf{refresh})$, we let $\Pr[A(m, \mathcal{H})^{I(\mathsf{PRG})} \to 1]$ denote the probability that $A$ outputs the bit "1" after interacting as above with the system. (Here $I(\mathsf{PRG})$ stands for the "ideal process" from above and note that we only use $\mathsf{PRG}$ in this game to answer queries that are made while the $\mathsf{compromised}$ flag is set to $\mathsf{true}$.)

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

DEFINITION 3.1. *We say that* $\mathsf{PRG} = (\mathsf{next}, \mathsf{refresh})$ *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*

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

*is negligible in the security parameter $m$.*

## 4. A 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 two components of our construction are the extraction function $\mathsf{extract}(\cdot)$ and a simple (non-robust) cryptographic PRG $\mathsf{G}(\cdot)$. A cryptographic PRG is a stateless function $\mathsf{G} : \{0,1\}^m \to \{0,1\}^{2m}$ 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 [3], or even any arbitrary one way function [9, 12].

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

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

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

Our main theorem regarding this pseudorandom generator is the following:

THEOREM 4.1. *Let $m$ be a security parameter, let* $\mathsf{extract} : \{0,1\}^{\geq m} \to \{0,1\}^m$ *and let $\mathcal{H}$ be a family of distributions over bit strings, such that* $\mathsf{extract}$ *is an $\mathcal{H}$-extractor. Also, let $\mathsf{G}$ 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 the real system $R(\mathsf{PRG})$.

**Expr. I** $A$ interacts with the ideal process $I(\mathsf{PRG})$.

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 $\mathsf{extract}(x)$ for $x \leftarrow_{\mathrm{R}} \mathcal{D} \in \mathcal{H}$). Namely, in **Expr. H** the attacker $A$ interacts with a modified process that works as follows:

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

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

$\mathsf{set\text{-}state}(s')$ with an $m$-bit string $s'$. The system returns to the attacker the current internal state $s$ and then changes it to $s'$.

$\mathsf{next\text{-}bits}()$. The system runs $(r, s') \leftarrow \mathsf{next}(s)$, 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 $\mathsf{good\text{-}refresh}(\cdot)$

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 $\mathsf{extract}(\cdot)$ in these $\mathsf{good\text{-}refresh}(\cdot)$ calls. In **Expr. H** the distribution over these strings is the uniform distribution, whereas in **Expr. R** each of these string is set to $\mathsf{extract}(x)$ where $x$ is chosen from (a convex combination of) the distribution in $\mathcal{H}$.[5] But since the statistical distance between the uniform distribution and $\mathsf{extract}(\mathcal{D})$ 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 $\mathsf{G}$ 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 $(r^*, s^*) \leftarrow \mathsf{G}(s)$ for a random $s \leftarrow_{\mathrm{R}} \{0,1\}^m$.

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 $i^* \leftarrow_{\mathrm{R}} \{1, 2, \dots, q\}$, and also setting $s \leftarrow 0^m$ and $\mathsf{compromised} \leftarrow \mathsf{true}$, 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:

$\mathsf{good\text{-}refresh}(\mathcal{D})$ with $\mathcal{D}$ a distribution in $\mathcal{H}$. If $i < i^*$ then $B$ chooses at random $s' \leftarrow_{\mathrm{R}} \{0,1\}^m$. If $i = i^*$ then $B$ uses its input, setting $s' = s^*$. If $i > i^*$ then $B$ chooses at random $d \leftarrow_{\mathrm{R}} \{0,1\}^m$ and sets $s' \leftarrow \mathsf{G}'(s \oplus d)$ where $s$ is the current internal state. Either way, $B$ updates the internal state to $s'$ and sets $\mathsf{compromised} \leftarrow \mathsf{false}$.

$\mathsf{bad\text{-}refresh}(x)$ with a bit string $x$. If $\mathsf{compromised} = \mathsf{true}$ or $i > i^*$ then $B$ sets $s' \leftarrow \mathsf{G}'(s \oplus \mathsf{extract}(x))$. If $\mathsf{compromised} = \mathsf{false}$ and $i < i^*$ then $B$ chooses at random $s' \leftarrow_{\mathrm{R}} \{0,1\}^m$, and if $\mathsf{compromised} = \mathsf{false}$ 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 $\mathsf{compromised}$).

$\mathsf{set\text{-}state}(s')$. $B$ returns to the attacker the current internal state $s$ and then changes it to $s'$, and also sets $\mathsf{compromised} \leftarrow \mathsf{true}$.

$\mathsf{next\text{-}bits}()$. If $\mathsf{compromised} = \mathsf{true}$ or $i > i^*$ then $B$ sets $(r, s') \leftarrow \mathsf{G}(s)$. If $\mathsf{compromised} = \mathsf{false}$ and $i < i^*$ then $B$ chooses at random $r, s' \leftarrow_{\mathrm{R}} \{0,1\}^m$, and if $\mathsf{compromised} = \mathsf{false}$ 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 $r$.

[5]Note 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 $\mathsf{good\text{-}refresh}(\cdot)$ call, so the overall distribution from which $x$ is drawn is a convex sum of all the distributions in $\mathcal{H}$.

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 $B$ 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 $\mathsf{compromised}$ 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 $\mathsf{PRG}$ when $\mathsf{compromised} = \mathsf{true}$ (in both the run of $B$ and in the experiment $H^{(0)}$). Also, all the queries $i < i^*$ with $\mathsf{compromised} = \mathsf{false}$ are answered by $B$ with random and independent bit strings, just as in the experiment $H^{(0)}$. (The only tricky case is a $\mathsf{set\text{-}state}(\cdot)$ 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 $\mathsf{compromised} = \mathsf{true}$.)

It is also clear that when $B$ chooses some $i^*$ and its input is chosen at random $(r^*, s^* \leftarrow_{\mathrm{R}} \{0,1\}^m)$ 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 $\mathsf{G}$ ($(r^*, s^*) \leftarrow \mathsf{G}(s)$ for $s \leftarrow_{\mathrm{R}} \{0,1\}^m$) 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 $\mathsf{good\text{-}refresh}(\cdot)$ then $B$ would set the internal state to $s' \leftarrow \mathsf{G}'(s)$ 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 $s' \leftarrow \mathsf{G}'(s \oplus d)$ with $s$ the prior state and a newly random choice $d \leftarrow_{\mathrm{R}} \{0,1\}^m$, so the distribution of $s'$ is the same in both.

- If the $i^*$ call of $A$ is $\mathsf{bad\text{-}refresh}(x)$ with $\mathsf{compromised} = \mathsf{false}$, then again $B$ would set the internal state to $s' \leftarrow \mathsf{G}'(s)$ 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 $s' \leftarrow \mathsf{G}'(s \oplus \mathsf{extract}(x))$ with $s$ the previous state. However, since $\mathsf{compromised} = \mathsf{false}$ and since all the calls upto $i^* - 1$ with $\mathsf{compromised} = \mathsf{false}$ 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 $\mathsf{next\text{-}bits}()$ with with $\mathsf{compromised} = \mathsf{false}$, then $B$ would set $(r, s') \leftarrow \mathsf{G}'(s)$ 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 $(r, s') \leftarrow \mathsf{G}'(s)$ with $s$ the previous state. Again, since $\mathsf{compromised} = \mathsf{false}$ and since all the calls upto $i^* - 1$ with $\mathsf{compromised} = \mathsf{false}$ 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 at most $\frac{p_0 - p_h}{q}$. $\square$

### 4.1 A concrete construction

Below we list some reasonable choices for implementing the cryptographic PRG and the randomness extractor. For the cryptographic PRG, there are many good implementations and the choice is rather arbitrary. One reasonable choice would be to use the AES block cipher in counter mode.

For the randomness extractor, on the other hand, the choice seems harder. In many cases it may be reasonable to use AES with a fixed random key in CBC mode. Here one may rely on the (partially heuristic) analysis of Dodis et al. [5], which we briefly describe in the appendix. The problem with that, however, is that plugging the block size $m = 128$ in the bounds from [5] gives a rather poor result. One could alternatively use HMAC-SHA1 for extraction, but again the result from [5] regarding HMAC-SHA1 are a bit too weak for comfort. Yet another alternative is to use a block cipher with block size of 256 bits (with a fixed random key and CBC mode) for the randomness extraction. A candidate cipher for this implementation is Rijndael, which has a variant with 256-bit blocks. (This last alternative would probably be our choice if we had to actually implement a robust generator.)

## 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 $\mathsf{good\text{-}refresh}(\mathcal{D})$. Most realistic implementations, however, would likely draw from their entropy sources frequently (e.g., every few milliseconds), but will only modify the internal state every so often (e.g., once every few minutes). 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, only keeping a few bytes (e.g., 256 bits) of state. As mentioned above, a plausible candidate for such an extractor is to use a block cipher with a fixed random key in CBC mode [5].

### 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 [13]). 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.[6] For example, when the system 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 every five minutes).

**Recommended refresh strategy.** 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 either using a periodic refresh with very low period (e.g., once every five minutes), or using a very low static estimate for the entropy (e.g., 1/2 entropy bit per sample). In the latter case, one should set a *minimum time interval* between consecutive automatic refreshes (e.g., not more than once a minute). This minimum delay is suggested to avoid an attack where an adversary generates a great number of events with zero entropy (as far as the adversary's knowledge is concerned). In any case, it is recommended to provide an interface to allow a *manual refresh* of the generator with user-supplied data (independently of the system-harvested entropy). The properties of our security definition guarantee that there is no damage in allowing an attacker the ability to perform such a manual refresh.

**The Fortuna heuristic.** The Fortuna architecture by Ferguson and Schneier [7] 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 (up to a factor of 64) 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 [7, Sec 10.5.2, Page 169]), so it 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 the 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

[6]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).

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 an "state-compromise extension attack" (a-la-[13]), and will only need to try two possible values for the refresh data to maintain its complete knowledge of all the pools. 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 used a single pool with a sufficiently 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. (Note that our model also assumes a conditional entropy requirement between all the good refreshes, since a good refresh is made from a distribution in $\mathcal{H}$ regardless of the outcome of the previous good refresh. However, it seems more reasonable to assume independence or conditional entropy between data that is collected in disjoint time intervals than between data that is collected in roughly the same time as in the case of Fortuna.)

**Refreshing after reboot/compromise.** One case where the "very rare refresh" rule of thumb cannot be applied is the initial refresh of the system when it is first started (or rebooted), or when it recovers from a known compromise. Clearly, in most cases we cannot stop the boot process for several minutes 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 where it only refreshes the state (say) every five minutes.[7] 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 is that after sampling data for an interval of $x$ seconds, the system would 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.

**Entropy testing.** Some works advocate the use of entropy estimators as a security measure, to avoid the generator "running on empty" (e.g., see [11]). As stated above, we reject this approach as entropy estimation in general is an inherently impossible task. Nevertheless, for some specific entropy sources (e.g., hardware-based generators) it may be possible to use tests to detect failures. In any case, we believe any such test should be tailored to the specific source and rightly considered part of the source sampling procedure and not part of the generator itself. In contrast to [11], we do not see any benefit in testing the *output* of the generator.

[7]Of 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.

## 5.3 Discussion and relevance to `/dev/random`

The current implementation of Linux has two sources of randomness: `/dev/random` that should provide information-theoretic (i.e. statistical) security, and `/dev/urandom` that should 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 SHA1 indeed provides information-theoretic security. Indeed, we suspect that it sometimes does not.

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 and holds even with respect to computationally unbounded attackers. In our view, there is nothing further from the truth. Leaving aside the fact that `/dev/random` uses SHA1 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 it is far safer to trust the security of a published cryptographic design such as HMAC-SHA1 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.

Moreover, we point out that `/dev/random` may fail to provide information-theoretic security *even if the entropy estimator is correct.* For example, in the version or `random.c` that was used in the Linux kernel v2.4, both streams used the same entropy pool, so the output of `/dev/urandom` leaked information also about the state of `/dev/random`. And even when the two streams use syntactically distinct pools (as in the Linux kernel v2.6), as long as they are refreshed from possibly dependent data there is no guarantee of information-theoretic security for `/dev/random`.

## 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.

## 6. CONCLUSIONS

In the course of this work we arrived at the following observations/recommendations for the design of robust pseudo-random generators.

**1.** The generator design should be explicitly split into two parts, namely the randomness extraction and the output generation. Moreover, it is possible to provide a modular design that allows changing the primitives to achieve different security/efficiency tradeoffs.

**2.** We advise against using run-time entropy estimation, and believe that having frequent automatic refreshes is almost never beneficial. Instead, we advocate making the refresh rate as low as possible, specifically in the order of once every few minutes.

**3.** In a well-designed generator, there is no harm in allowing refresh data to come from any source, even one that is not trusted. This can be important, e.g., for smartcards, or when allowing user-supplied data to be used for refreshing the state.

## 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. Finally, we thank the ACM-CCS reviewers for many useful comments.

## 7. REFERENCES

[1] B. Barak, R. Shaltiel, and E. 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.

[2] M. Bellare and B. Yee. Forward-security in private-key cryptography. In *Topics in Cryptology - CT-RSA'03*, pages 1–18, 2003.

[3] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, Nov. 1984. Preliminary version in FOCS '82.

[4] R. Canetti and A. Herzberg. Maintaining security in the presence of transient faults. In *Crypto '94*, pages 425–438, 1994. LNCS No. 839.

[5] Y. Dodis, R. Gennaro, J. Håstad, H. Krawczyk, and T. Rabin. Randomness extraction and key derivation using the CBC, Cascade and HMAC modes. In *Crypto '04*, pages 494–510, 2004. LNCS No. 3152

[6] Y. Dodis and A. Smith. Entropic security and the encryption of high entropy messages. In *Theory of Cryptography Conference (TCC) '05*, pages 556–577, 2005.

[7] N. Ferguson and B. Schneier. *Practical Cryptography.* Wiley, New York, NY, USA, 2003.

[8] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal*, pages 66–70, 1996.

[9] O. Goldreich and L. A. Levin. A hard-core predicate for all one-way functions. In *Proc. 21st STOC*, pages 25–32. ACM, 1989.

[10] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proc. 19th STOC*, pages 218–229. ACM, 25–27 May 1987.

[11] P. Gutmann. Software generation of practically strong random numbers. In *Proceedings of the 7th USENIX Security Symposium*, 1998. Available from `http://www.cs.auckland.ac.nz/~pgut001/`.

[12] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999. Preliminary versions appeared in STOC' 89 and STOC' 90.

[13] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudorandom number generators. In *FSE '98*, pages 168–188, 1998. LNCS No. 1372.

[14] R. Shaltiel. Recent developments in extractors. In G. Paun, R. I. Virgili, G. Rozenberg, and A. Salomaa, editors, *Current trends in theoretical computer science.*, volume 1. World Scientific, 2004. Preliminary version in bulletin of the EATCS, 2002. Available on `http://www.cs.haifa.ac.il/~ronen/`.

## 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 $\mathsf{extract} : \{0,1\}^{\geq m} \to \{0,1\}^m$ 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 $\mathsf{extract}$ 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. [1] that sets up a model obtaining essentially the most general family $\mathcal{H}$ and shows a rater efficient combinatorial construction that can be proven to work in that model. Later we describes briefly the work of Dodis et al. [5] that deals with using constructions such as CBC-AES and HMAC-SHA1 for extraction.

### A.1 The Barak-Shaltiel-Tromer model

The work [1] deals with extracting truly random bits from sources that have high entropy but are far from uniform. The goal is to construct $\mathcal{H}$-extractors (cf. Definition 2.1) where $\mathcal{H}$ includes 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.[8] Hence, we require that all the distributions in $\mathcal{H}$ will have more than $m$ bits of min-entropy.

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-entropy[9], 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 [1] utilizes *randomized* extraction functions, $\mathsf{extract} : \mathsf{Coins} \times \{0,1\}^n \to \{0,1\}^m$, (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,
$$\mathcal{H} = \{\mathcal{D}_1, \ldots, \mathcal{D}_{2^t}\}$$
2. The designer chooses "once and for all" the randomness $s \leftarrow_{\mathrm{R}} \mathsf{Coins}$ for the extraction function. The coins $s$ are made public (and in particular given to the attacker).

The randomized construction $\mathsf{extract}$ is considered good if for every family $\mathcal{H}$ of $2^t$ distributions, for all but a $2^{-m}$ fraction of the coins $s \in \mathsf{Coins}$, the deterministic function $\mathsf{extract}_s(\cdot) = \mathsf{extract}(s, \cdot)$ is an *$\mathcal{H}$-extractor* as per Definition 2.1. Namely, no matter how the attacker chooses its family $\mathcal{H}$ in Step 1, the resulting extraction function $\mathsf{extract}_s$ 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 shown in [1] how to construct efficient randomized extraction functions $\mathsf{extract}$ 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 [1] 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, we do not claim that the model in [1] 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 Using cryptographic modes of operation

Many systems today use cryptographic functions in some mode of operation for randomness extraction. 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 [5]. Specifically, they studied the CBC-$\Pi$ construction with $\Pi$ 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-$\Pi$ for a random permutation $\Pi$ 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 $\Pi$, 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 $y$, $\Pr_{x \leftarrow_{\mathrm{R}} \mathcal{D}}[\text{CBC-}\Pi(x) = y] = 2^{-m}(1 \pm \frac{L}{2^{m/2}})$. Their results for HMAC-$f$ are somewhat weaker (but similar in spirit).

[8] A closely related measure is the Renyi entropy of order two.

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