True Random Number Generators Secure in a Changing Environment

Boaz Barak, Ronen Shaltiel, and Eran Tromer

Department of Computer Science and Applied Mathematics
Weizmann Institute of Science , Rehovot, ISRAEL
Email: {boaz,ronens,tromer}@wisdom.weizmann.ac.il

Abstract. A true random number generator (TRNG) usually consists of two components: an "unpredictable" source with high entropy, and a *randomness extractor* — a function which, when applied to the source, produces a result that is statistically close to the uniform distribution.
When the output of a TRNG is used for cryptographic needs, it is prudent to assume that an adversary may have some (limited) influence on the distribution of the high-entropy source. In this work:

1. We define a mathematical model for the adversary's influence on the source.
2. We show a simple and efficient randomness extractor and *prove* that it works for *all* sources of sufficiently high-entropy, even if individual bits in the source are correlated.
3. Security is guaranteed even if an adversary has (bounded) influence on the source.

Our approach is based on a related notion of "randomness extraction" which emerged in complexity theory. We stress that the statistical randomness of our extractor's output is *proven*, and is not based on any unproven assumptions, such as the security of cryptographic hash functions.
A sample implementation of our extractor and additional details can be found at a dedicated web page [Web].

1 Introduction

1.1 General setting

It is well known that randomness is essential for cryptography. Cryptographic schemes are usually designed under the assumption of availability of an endless stream of unbiased and uncorrelated random bits. However, it is not easy to obtain such a stream. If not done properly, this may turn out to be the Achilles heel of an otherwise secure system (e.g., Goldberg and Wagner's attack on the Netscape SSL implementation [GW96]).

In this work we focus on generating a stream of truly random bits. This is the problem of constructing a *true random number generator* (TRNG). The usual way to construct such a generator consists of two components:

1. The first component is a device that obtains some digital data that is unpredictable in the sense that it has high entropy.[1] This data might come from various sources, such as hardware devices based on thermal noise or radioactive decay, a user's keyboard typing pattern, or timing data from the hard disk or network. We stress that we only assume that this data has high entropy. In particular, we do *not* assume that it has some nice structure (such as independence between individual bits). We call the distribution that is the result of the first component the *high-entropy source*.
2. The second component is a function, called here a *randomness extractor*, which is applied to the high-entropy source in order to obtain an output string that is shorter, but is random in the sense that it is distributed according to the uniform distribution (or a distribution that is statistically very close to the uniform distribution).

Our focus is on the second component. The goal of this work is to construct a single extractor which can be used with *all* types of high-entropy sources, and that can be *proven* to work, even in a model that allows an adversary some control over the source.

Running a TRNG in adversarial environments. The high entropy source used in a TRNG can usually be influenced by changes in the physical environment of the device. These changes can include changes in the temperature, changes in the voltage or frequency of the power supply, exposure to radiation, etc.. In addition to natural changes in the physical environment, if we are using the output of a TRNG for cryptographic purposes, it is prudent to assume that an adversary may be able to control at least some of these parameters. Of course, if the adversary can have enough control over the source to ensure that it has zero entropy then, regardless of the extractor function used, the TRNG will be completely insecure. However, a reasonable assumption is that the adversary has only partial control over the source in a way that he can influence the source's output, but not remove its entropy completely.

1.2 Our results

In this paper, we suggest a very general model which captures such adversarial changes in the environment and show how to design a randomness extractor that will be secure even under such attacks.

In all previous designs we are aware of, either there is no mathematical treatment or the source of random noise is assumed to have a nice mathematical structure (such as independence between individual samples). As the nature of cryptanalytic attacks cannot be foreseen in advance, it is hard to be convinced

[1] Actually, the correct measure to consider here is not the standard Shannon entropy, but rather the measure of *"Min-Entropy"* (see Remark 1). In the remainder of the paper we will use the word "entropy" loosely as a measure of the amount of randomness in a probability distribution.

of the security of a TRNG based on a set of statistical tests that were performed on a prototype in ideal conditions. We also remark that it may be dangerous to assume that the source of randomness has a nice mathematical structure, especially if the environment in which the TRNG operates may be altered by an adversary.

Our extractor is simple and efficient, and compares well with previous designs. It is based on pairwise-independent hash function [WC81].[2] Our approach is inspired by a somewhat different notion of "randomness extractors" defined in complexity theory (see surveys [NTS99,Sha02] and Section 1.3).

Our design works in two phases:

Preprocessing: In this phase the manufacturer (or the user) chooses a string π which we call a *public parameter*. This string is then hardwired into the implementation and need not be kept secret. The *same* string π can be distributed in all copies of the randomness extractor device, and will be used whenever they are executed. (We discuss this in detail in Section 1.4).

Runtime: In this phase the randomness extractor gets data from the high-entropy source and its output is a function of this data and the public parameter π.

The analysis guarantees that if π is chosen appropriately in the preprocessing phase and the high-entropy source has sufficient entropy then the output of the TRNG is essentially uniformly distributed even when the environment in which the TRNG operates is altered by an adversary. This guarantee holds as long as the adversary has limited influence on the high-entropy source.

In particular, we make no assumption on the *structure* of the high-entropy distribution except for the necessary assumption that it contains sufficient entropy. Existing designs of high-entropy sources seem to achieve this goal.

1.3 Previous works

Randomness extractors used in practice. As far as we are aware, all extractors previously used in practice as a component in a TRNG, fall under the following two categories:

Designs assuming mathematical structure These are extractors that work under the assumption that the physical source has some "nice" mathematical structure.

An example of such an extractor is the *von Neumann extractor* [vN51], used in the design of the Intel TRNG [JK99]. On input a source $X_1, \ldots, X_n$ the von Neumann extractor considers successive pairs of the form X_{2i}, X_{2i+1}; for each pair, if $X_{2i} \neq X_{2i+1}$ then X_{2i} is sent to the output, otherwise nothing it sent. The von Neumann extractor works if one assumes that the all bits in the source are independent and are identically distributed. That is, each

[2] Some choices of the parameters require use of ℓ-wise independent hash functions for $\ell > 2$.

bit in the source will be equal to 1 with the same probability p, and this will happen independently of the values of the other bits. However, it may fail if different bits are correlated or have different biases.

Other constructions that are sometimes used have every bit of the output be XOR of bits in the source that are "far from each other". Such constructions assume that these "far away" bits are independent.

RFC 1750 [ErCS94] also suggests some heuristics such as applying a Fast Fourier Transform (FFT) or a compression function to the source. However, we are not aware of any analysis of the conditions on the source under which these heuristic will provide a uniform output.

Applying a cryptographic hash function Another common approach (e.g., [ErCS94], [Zim95]) is to extract the randomness by a applying a cryptographic hash function (or a block cipher) to the high-entropy source. The result is expected to be a true random (or at least *pseudo*-random) output. As there is no mathematical guarantee of security, confidence that such constructions work comes from the extensive cryptanalytic research that has been done on these hash function. However, this research has mostly been concentrated on specific "pseudorandom" properties (e.g., collision-resistance) of these functions. It is not clear whether this research applies to the behavior of such hash functions on sources where the only guarantee is high entropy, especially when these sources may be influenced by an adversary that knows the exact hash function that is used.

Randomness extractors in complexity theory The problem of extracting randomness from high-entropy distributions is also considered in complexity theory (for surveys, see [NTS99,Sha02]). However, the model considered there allows the adversary to have full control over the source distribution. The sole restriction is that the source distribution has high entropy. One pays a heavy price for this generality: it is impossible to extract randomness by a *deterministic* randomness extractor.[3] Consequently, this notion of randomness extractors (defined in [NZ96]) allows the extractor to also use few additional truly random bits. The rationale is that the extractor will output many more random bits than initially spent. While this concept proves to be very useful in many areas of computer science, it does not provide a way to generate truly random bits for cryptographic applications.[4]

Nevertheless, our solution uses techniques from this area. For the reader familiar with this area, we remark that our solution builds on observing that a weaker notion of security (the one described in this paper) can be

[3] Consider even the simpler task of extracting a single bit. Every candidate randomness extractor $E : \{0,1\}^n \to \{0,1\}$ partitions $\{0,1\}^n$ into two sets B_0, B_1 where B_i is the set of all strings mapped to i by E. Assume w.l.o.g. that $|B_0| \geq |B_1$. Then the adversary can choose the source distribution X to be the uniform distribution over B_0 and thus, $E(X)$ is always fixed as 0 and is not at all random. Note that if $E(\cdot)$ can be computed efficiently, then this distribution X can also be sampled efficiently.

[4] Some weaker notions of randomness extractors were proposed. These notions usually suggest considering restricted classes of random sources. See [TV02] and the references there.

guaranteed even when the few additional random bits are chosen once and for all by the manufacturer and made public.

1.4 Advantages and disadvantages of our scheme.

The main advantage of our scheme is that it is *proven* to work for *every* high-entropy source, provided that the adversary has only limited control on the distribution of the source. By contrast, previous schemes are either known to fail for some very natural high-entropy sources (e.g., the von Neumann's extractor), or lack a relevant formal analysis (see above).

Efficiency. It is natural to measure the performance of a randomness extractor in terms of the cost per output bit. This measure depends on the following factors:

1. *Cost:* The speed and size of the hardware or software implementation of the extractor.
2. *Entropy rate:* The amount of entropy contained in the source.
3. *Entropy loss:* The difference between the amount of entropy that the high-entropy source contains and the number of bits extracted.

Our design allows tuning the running time and entropy loss as a function of the expected entropy rate and the desired resiliency against adversarial effects on the source. This tuning helps to achieve good overall performance in different scenarios. We discuss specific scenarios below.

In general, our approach is quite simple and efficient and is suitable for a hardware implementation. Its cost is comparable to that of cryptographic hash functions, and it can provably achieve low entropy loss and extract more than half of the entropy present in the source (by comparison, the von Neumann extractor extracts at most half of the entropy)[5].

Example: low entropy rate. For example, consider the case where the source is the typing patterns of a user. In this case the speed at which one can sample the high-entropy source is comparatively slow, and furthermore sampling the source may be expensive. It is thus crucial to minimize entropy loss and extract as much as possible from the entropy present in the source. Our design allows extracting 3/4 of the entropy in the source at a slight cost to the running time. In this case, the running time is less significant as the bottleneck is the sampling speed from the random source.

Example: high entropy rate. Consider the case where the source is sampling of thermal noise. Now the running time is important and we can tune our design to work faster at the cost of higher entropy loss.

The existence of a formal proof of security can be helpful when optimizing the implementation. Our proof shows that any implementation of "universal hash

[5] The basic von Neumann extractor can be extended to extract more bits at some cost to the algorithm's efficiency [Per92].

functions" (or "ℓ-wise independent hash functions") suffices for our randomness extractor. Thus, a designer can choose the most efficient implementation he finds and optimize it to suit his particular architecture. This is contrast to cryptographic hash functions, which do not have a proof of security and where the effect of changes (e.g., removing a round) is unknown, and thus such optimizations are not recommended in practice.

A public parameter. One *disadvantage* of our scheme is the fact that it uses a *public parameter*.[6] The security of the scheme is proven under the assumption that the parameter is chosen at random. This parameter needs to be chosen only once and the resulting scheme will be secure with extremely high probability.

We stress that we do *not* assume that this parameter is kept secret. Moreover, this parameter can be chosen once and for all by the manufacturer and hardwired into all copies of the device. We also do not assume that the distribution of the high-entropy source is completely independent from the choice of this parameter — our model allows this distribution to be partially controlled by a computationally-unbounded adversary that knows the public parameter.

Note that a public parameter is *necessary* to obtain the security properties that we require.

2 The formal model

2.1 Preliminaries

Min-Entropy. The *min-entropy* of the source X, denoted by min-Ent(X), the maximal number k such that for every $x \in X$, $\Pr[X = x] \leq 2^{-k}$.

Remark 1. Min-entropy is a stricter notion than the standard (Shannon) entropy, in the sense that the min-entropy of X is always smaller than or equal to the Shannon entropy of X.

It is easy to see that it is impossible to extract m bits from a distribution X with min-Ent$(X) \leq m - 1$. This is because such a distribution gives probability at least $2^{-(m-1)}$ to some element x. It follows that for any candidate extractor function $E : \{0,1\}^n \to \{0,1\}^m$ the element $y = E(x)$ has probability at least $2^{-(m-1)}$ and thus $E(X)$ is far from being uniformly distributed.

We conclude that having min-entropy larger than m is a *necessary* condition for randomness extraction. In this paper we show that having min-entropy k slightly larger than m is a sufficient condition.

Statistical Distance. We use dist(X, Y) to denote the *statistical distance* between X and Y that is: $\frac{1}{2}\sum_a |\Pr[X = a] - \Pr[Y = a]|$. We say that X is *ϵ-close* to Y if dist$(X, Y) < \epsilon$.

[6] The description of many hash functions and block ciphers includes various semi-arbitrary constants; arguably these can also be considered public parameters.

Table 1. List of parameters

n	The length (in bits) of a sample from the high-entropy source.
k	The min-entropy of the high-entropy source.
t	The adversary can alter the environment in at most 2^t different ways
m	The length (in bits) of the output of the randomness extractor.
ϵ	The statistical distance between the uniform distribution and the output of the randomness extractor.

Notation for probability distributions. We denote by U_m the uniform distribution on strings of length m. If X is a distribution then by $x \in_R X$ we mean that x is chosen at random according to the distribution X. If X is a set then we mean that x is chosen according to the uniform distribution on the set X. If X is a distribution and $f(\cdot)$ is a function, then we denote by $f(X)$ the random variable that is the result of choosing $x \in_R X$ and computing $f(x)$.

2.2 The parameters

The parameters for our design are listed in Table 1. We think of samples from the source as coming in blocks of n bits.

The goal is to design an extractor that, given an adversarially-chosen n-bit source with k bits of entropy, is resilient against as much adversary influence as possible (i.e., maximize t), while extracting as many random bits as possible (i.e, maximize m), with negligible statistical distance ϵ. In Section 2.5 we give a sample setting of the parameters.

2.3 Definition of security

Definition 1 (Extractor). *An* extractor *is a function* $E : \{0,1\}^n \times S \to \{0,1\}^m$ *for some set* S*.*

Denote by $E^\pi(\cdot) = E(\cdot, \pi)$ the one-input function that is the result of fixing the parameter π to the extractor E. We would like the output $E(X, \pi) = E^\pi(X)$ to be (close to) uniformly distributed, where $X \in \{0,1\}^n$ is the output of the high-entropy source and $\pi \in S$ is the public parameter.

Defining Security. We consider the following ideal setting:

1. An adversary chooses 2^t distributions $\mathcal{D}_1, \ldots, \mathcal{D}_{2^t}$ over $\{0,1\}^n$, such that min-Ent$(\mathcal{D}_i) > k$ for all $i = 1, \ldots, 2^t$.
2. A public parameter π is chosen at random and independently of the choices of $\mathcal{D}_i$.
3. The adversary is given π, and selects $i \in \{1, \ldots, 2^t\}$.
4. The user computes $E^\pi(X)$, where X is drawn from $\mathcal{D}_i$.

Definition 2 (t-resilient extractor). *Given n, k, m, ϵ and t, an extractor E is* t-resilient *if, in the above setting, with probability $1 - \epsilon$ over the choice of the public parameter the statistical distance between $E^\pi(X)$ and U_m is at most ϵ.*

Interpretation. The above ideal setting is intended to capture security in the following scenario. A manufacturer designs a noise generating device whose output is a random variable X. Ideally, we would like the adversary not to be able to influence the distribution of X at all. However, in a realistic setting the adversary has some control over the environment in which the device operates (temperature, voltage, frequency, timing, etc.), and it is possible that that changes in this environment affect the distribution of X. We assume that the adversary can control at most t boolean properties of the environment, and can thus create at most 2^t different environments. The user observes the value of X which, conditioned on the choice of environment being i, is distributed as $\mathcal{D}_i$. The definition of security guarantees that the output of the extractor is close to uniformly distributed.

In fact, the security definition (which our construction fulfills) may be stronger than necessary in several senses:

- We do not assume any computational bound on the adversary.
- We do not assume that the user knows which environment i was chosen.
- More fundamentally, we do not require either the user nor the manufacturer need to knows which properties of the environment are controllable by the adversary. The only limitation is that the adversary can control at most t (boolean) properties, and that the source entropy is at least k.
- We allow adversarial choice of all the source distribution for each of the 2^t environment settings. Thus, even for "normal" environment settings, the source may behave in the worst possible way subject to the above requirements. By contrast, in the real world the source distributions would be determined by the manufacturer, presumably in the most favorable way.
- The behavior of the source may change arbitrarily for different environments (i.e., the distributions $\mathcal{D}_i, \mathcal{D}_j$ for $i \neq j$ need not be related in any way). In the real world, many properties of the source would persist for all but the most extreme environment settings.

Remark 2. One can make a stricter security requirement by allowing the adversary to choose i not only as a single value, but also as a random variable with an arbitrary distribution over $\{1, \ldots, 2^t\}$. The two definitions are equivalent.

Remark 3. Many applications require a long stream of output bits. In such cases, our extractor can simply be applied to successive blocks of inputs, always with the same fixed public parameter. The security is guaranteed as long as each input block contains k bits of *conditional min-entropy* (defined analogously to conditional entropy), conditioned on all previous blocks. Of course, this still requires that the conditional distribution of every input block is one of $\mathcal{D}_1, \ldots, \mathcal{D}_{2^t}$.

2.4 Our result

Our main result is an efficient design for a t-resilient extractor for a wide range of the parameters. We have the following theorem:

Theorem 1. *For every n, k, m and ϵ there is a t-resilient extractor with a public parameter of length $2n$ such that* [7]

$$t = \frac{k-m}{2} - 2\log(1/\epsilon) - 1$$

We can increase t at the cost of an increase in the running time and the length of the public parameter π, to obtain:

Theorem 2. *For every n, k, m and ϵ and $\ell \geq 2$ there is a t-resilient extractor with a public parameter of length ℓn such that*

$$t = \frac{\ell}{2}(k - m - 2\log(1/\epsilon) - \log \ell + 2) - m - 2 - \log(1/\epsilon)$$

We explain our construction in Section 3 and prove its security in Appendix A. In Section 4 we present potential implementations.

Note that in the above theorems, t does not depend on n. In other words, the resiliency t depends only on the amount of of entropy loss (i.e., $k - m$), the statistical distance (i.e., ϵ) that we are willing to accept and the parameter ℓ.

2.5 Sample settings of parameters

As the theorems involve many parameters we give concrete examples for natural choices. In the following examples we will consider extracting $m = 256$ bits which are $\epsilon = 2^{-35}$-close to uniform from a source containing $k = 512$ bits of entropy. The choice of n (the length of the source) should depend on the expected quality of the random sample. For example, if the source is the typing pattern of a user then its entropy rate is low and we may need to set $n \approx 2500$ in order to have 512 bits of entropy, while for dedicated noise-generation hardware we may assume that the entropy rate is very high and set $n = 768$.

Using Theorem 1 we get $t = 57$ for $k = 512$. Using the less efficient design of Theorem 2 we can improve both entropy loss and security: choosing $\ell = 16$ we get $t = 667$ and can even reduce k to $k = 448$. These numbers are just an illustration and different tradeoffs between performance, entropy loss and guarantee of security can be made.

3 Our Design

In this section, we present our construction which is based on the notion of "ℓ-wise independent hash functions". We formally define this notion and describe our construction in these term in Section 3.1 and prove correctness in Appendix A. We discuss implementation of this construction in Section 4.

[7] In fact, the constructions described in Section 4 have a shorter public parameter, of length n and $n + m - 1$.

3.1 Randomness extractors from ℓ-wise independence

We start by recalling the notion of ℓ-wise independence.

Definition 3 (ℓ-wise independence). *A collection $Z_1, \cdots, Z_n$ of random variables is called ℓ-wise independent if for every $i_1, \cdots, i_\ell \in \{1, \cdots, n\}$ the random variables $Z_{i_1}, \cdots, Z_{i_\ell}$ are independent.*

A very useful tool is the notion of ℓ-wise independent families of hash functions. Intuitively, such functions have some properties of random functions even though they're much less random.

Definition 4 (ℓ-wise independent families of hash functions). *Given a collection $H = \{h_s\}_{s \in S}$ of functions $h_s : \{0,1\}^n \to \{0,1\}^m$, we consider the probability space of choosing $s \in_R S$. For every $x \in \{0,1\}^n$ we define the random variable $R_x = h_s(x)$ (Note that x is fixed and s is chosen at random). We say that H is an ℓ-wise independent family of hash functions if:*

- *For every x, R_x is uniformly distributed in $\{0,1\}^m$.*
- *The random variables $\{R_x\}_{x \in \{0,1\}^n}$ are ℓ-wise independent.*

The usefulness of this definition stems from the fact that there are such families which are relatively small (of size $2^{\ell n}$) and for which $h_s(x)$ can be efficiently computed given s and x.

In these terms, our construction is described as follows: randomly choose $s \in_R S$ (this is the "public parameter"), and let the randomness extractor be simply

$$E(x) = h_s(x)$$

In Appendix A we show that for appropriate parameters, this yields a t-resilient extractor. The following section describes concrete constructions.

4 Implementation

This section describes several extractor implementations based on known constructions of pairwise-independent hash functions [CW79,WC81]. Recall that an implementation of our randomness extractor is constructed has two phases:

- Preprocessing: Choosing the public parameter π.
- Runtime: Running $E^\pi(x)$ on a given string x.

As the first phase is done once and for all at a preprocessing phase, we focus on optimizing the resources used by the second phase. Also, the known implementations of ℓ-wise independent hash functions (for large ℓ) have higher implementation cost than 2-wise independent hash functions; we thus consider only implementations of 2-wise independent hash functions.

4.1 Linear functions

Theorem 3. *Let $GF(2^n)$ be the field with 2^n elements, and $S = \{(a,b)|a,b \in GF(2^n)\}$. For $s = (a,b)$ and $m < n$ define: $h_s(x) = (a \cdot x + b)_{1,\cdots,m}$ (i.e., the first m bits of $a \cdot x + b$ where arithmetic operations are in $GF(2^n)$). Then the family $H_{n,m} = \{h_s\}_{s \in S}$ is a 2-wise independent family of hash functions.*

The field $GF(2^n)$ can be realized as as the field of polynomials over $GF(2)$ modulo an irredicible polynomial of degree n. When the field element are represented as coefficient vectors, addition is the bitwise XOR operation while multiplication requires a modular reduction operation that is easily implemented in hardware for small n, but grows expensive for larger ones and is not well suited for software implementations.

In Appendix A we show that for every random variable X with min-Ent$(X) \geq k$, for an appropriate choice of m we have that for most pairs $s = (a,b)$, $h_s(X)$ is close to uniform.

We have $s = (a,b)$, $h_s(x) = (ax)_{1,\cdots,m} \oplus b_{1,\cdots,m}$. Define $g_a(x) = (ax)_{1,\cdots,m}$. For every fixed pair (a,b), $b_{1,\cdots,m}$ is constant. Thus, $h_{(a,b)}(X)$ is close to uniform if and only if $g_a(X)$ is close to uniform. This yields the following extractor for any $m < n$, where the relation to t and ϵ follows from Theorem 1.

- Preprocessing: choose some irreducible polynomial of degree n. Choose a string $a \in \{0,1\}^n$ at random and set $\pi = a$.
- Runtime: $E^\pi(x) = (a \cdot x)_{1,\cdots,m}$, using multiplication in $GF(2^n)$.

Remark 4. A family of ℓ-wise independent hash functions can be constructed in a similar manner by setting $s = (a_1, \cdots, a_\ell)$ and $h_s(x) = \sum_{1 \leq i \leq l} a_i \cdot x^{i-1}$.

4.2 Binary Toeplitz matrices

Theorem 4. *Let $\mathcal{F}$ be a finite field and let n', m' be integers, $m' < n'$. Let $S = \mathcal{F}^{n'+m'-1}$. For $s \in S$ and $x \in \mathcal{F}^n$, define $h_s(x) \in \mathcal{F}^{m'}$ by $(h_s(x) \in \mathcal{F}^{m'})_i = \sum_{j=0}^{n} x_i s_{i+j}$. Then the family $H_{n',m'} = \{h_s\}_{s \in S}$ is a 2-wise independent family of hash functions.*

The function $h_s(x)$ can be thought of as multiplication of an $m \times n$ Toeplitz matrix (whose diagonals are given by s) by the vector x. Alternatively, it can be considered a convolution of x and y. For $\mathcal{F} = GF(2)$, $n' = n$, $m' = m$, $h_s(x)$ we get the following extractor any $m < n$, (as before, t and ϵ follows from Corollary 1).

- Preprocessing: Choose a string $\pi \in \{0,1\}^{n+m-1}$ at random.
- Runtime: m bits such that the i-th bit is $\bigoplus_{i=0}^{n}(x_i \wedge s_{i+j})$.

For reasonable n and m, this can be implemented very efficiently in hardware, though in software the bit operations are somewhat inconvenient. To evaluate this, we tested a software implementation of this construction, written in plain C. For the parameters $n = 768$, $m = 256$ of Section 2.5, this implementation had a throughput of 36Mbit/sec (measured at the input) when executed on a 1.7GHz Pentium Xeon processor (cf. [Web]).

4.3 Toeplitz matrices over $GF(2^k)$

Since Theorem 4 applies to any finite field, we may benefit from using $GF(2^k)$ for $k > 2$. For example, consider $\mathcal{F} = \mathcal{F}'[x]/(x^2 + rx + 1) \cong GF(2^{16})$ where $\mathcal{F}' = GF(2)[x']/(x'^8 + x'^4 + x'^3 + x' + 1) \cong GF(2^8)$ and $r = x' \in \mathcal{F}'$ (both modulus polynomials are irreducible over the respective fields; the latter is taken from the Rijndael cipher.) Using this field, a direct implementation of the convolution performs just $n/16$ field multiplications for every 16 output bits.

The fields $\mathcal{F}, \mathcal{F}'$ are very suitable for software implementation, as follows. For $a, b \in \mathcal{F}' \setminus \{0\}$, multiplication in $\mathcal{F}'$ can be realized as $ab = \exp(\log_z a + \log_z b)$, where z is some generator of $\mathcal{F}'$; $\exp_z$ and $\log_z$ can be implemented via two small lookup tables [Gla02]. Multiplication in $\mathcal{F}$ can then realized by 5 multiplications in $\mathcal{F}'$:

$$(a_1 x + a_0) \cdot (b_1 + x b_0) \equiv a_0 b_0 - c + (a_0 b_1 + a_1 b_0 - rc)x \quad \text{over } \mathcal{F}$$

where $c = a_1 b_1$ and $r = x' \in \mathcal{F}'$ is a constant (02_h as a bit vector). All additions can be done as bitwise XOR, so overall each multiplication over $\mathcal{F}$ requires 4 XOR operations, 5 integer additions, a few shifts, $5 \cdot 3$ table lookups (into the L1 cache) and $5 \cdot 2$ tests whether $a, b = 0$. [8] A straightforward C implementation of the above description achieved a throughput of 56Mbit/sec in the same settings as above (cf. [Web]).

4.4 Randomness tests

We subjected the above implementations to the DIEHARD suite of statistical tests [Mar95]. For the seed, we used 1023 truly random bit generated by the `/dev/random` TRNG of Linux 2.4.20. For the source, we generated a 90MB file of English text by retrieving a large number of e-texts from Project Guttenberg, discarding the first 1000 lines of each file (this contains a common header) and eliminating all whitespaces. Thus the source data included the texts of Moby Dick and Frankenstein, the complete works of Shakespeare, and other such "random" data. We then executed the extractor on successive blocks of n bits, with $n = 768$ and $m = 256$, to get 30MB of output bits. The DIEHARD tests did not detect any anomaly in this output.[9] The test report is available at [Web].

5 Conclusions

In this work, we provide an extractor function that is *proven* to work in a model that allows for some adversarial influence on the high-entropy source. The most obvious question is whether the real world conditions satisfy the assumptions

[8] The multiplication of c by the constant r can be computed by a single lookup; this eliminates 1 addition, 2 lookups and 2 tests.

[9] In fact, on first attempt DIEHARD did report certain anomalies in one test. A careful inspection revealed that our source data accidentally included several nearly-identical versions of some literary works.

of our model. For example, suppose that a manufacturer constructs a device that outputs a distribution with min entropy at least k in the "benign" (i.e., non-adversarial) settings. Suppose now that he wants to apply our extractor to the output of this device, in an environment that may be somewhat influenced by the adversary.

One concern the manufacturer may have is how to ensure that under all possible adversarial influences, the entropy of the source will remain sufficient? This is indeed a valid concern, but if this is not met then the result will be insecure regardless of the extractor function used (since the adversary will be able to reduce the source's entropy, and no extractor function can *add* entropy). Therefore, it is the responsibility of the manufacturer to make sure that the device satisfies this condition. When this is fulfilled, our construction gives explicit guarantees on the quality of the extractor's output.

We stress that while the manufacturer still needs to carefully design the high-entropy source to be as independent as possible from environmental influence, the overall scheme will work even if design is not perfect and the adversary can affect the source in unpredictable ways, subject to the constraints assumed in our security model.

Acknowledgements

We thank Moni Naor and Adi Shamir for helpful discussions, and the anonymous referees for their constructive comments.

References

[BR94] M. Bellare and J. Rompel. Randomness-efficient oblivious sampling. In *35th Annual Symposium on Foundations of Computer Science*, 1994.

[CW79] L. Carter and M. Wegman. Universal hash functions. *JCSS: Journal of Computer and System Sciences*, 18:143–154, 1979.

[ErCS94] D. Eastlake, 3rd, S. Crocker, and J. Schiller. RFC 1750: Randomness recommendations for security, December 1994.

[Gla02] Brian Gladman. A specification for Rijndael, the AES algorithm. Available from `http://fp.gladman.plus.com/cryptography_technology/rijndael/aesspec.pdf` , 2002.

[GW96] Ian Goldberg and David Wagner. Randomness and the netscape browser. *Dr. Dobb's Journal*, pages 66–70, 1996.

[ILL89] R. Impagliazzo, L.A. Levin, and M. Luby. Pseudorandom generation from one-way functions. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, 1989.

[JK99] Benjamin Jun and Paul Kocher. The Intel random number generator. Technical report, Cryptography Research Inc., 1999. Available from `http://www.intel.com/design/security/rng/rngppr.htm` .

[Mar95] George Marsaglia. DIEHARD, a battery of tests for random number generators. Available from `http://stat.fsu.edu/~geo/diehard.html` , 1995.

[NTS99] Nisan and Ta-Shma. Extracting randomness: A survey and new constructions. *JCSS: Journal of Computer and System Sciences*, 58, 1999.

[NZ96] Noam Nisan and David Zuckerman. Randomness is linear in space. *Journal of Computer and System Sciences*, 52(1):43–52, February 1996.

[Per92] Yuval Peres. Iterating von Neumann's procedure for extracting random bits. *Ann. Statist.*, 20(1):590–597, 1992.

[Sha02] Ronen Shaltiel. Recent developments in extractors. *Bulletin of the European Association for Theoretical Computer Science*, 77, 2002.

[TV02] L. Trevisan and L. Vadhan. Pseudorandomness and average-case complexity via uniform reductions. In *Proceedings of the 17th Annual Conference on Computational Complexity*, 2002.

[vN51] John von Neumann. Various techniques used in connection with random digits. *Applied Math Series*, 12:36–38, 1951.

[WC81] M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *JCSS: Journal of Computer and System Sciences*, 22, 1981.

[Web] Web page for this paper. Available from `http://www.wisdom.weizmann.ac.il/~tromer/trng/` .

[Zim95] Philip R. Zimmermann. *PGP: Source Code and Internals*. MIT Press, Cambridge, MA, USA, 1995.

A Proof of the Main Theorems

We begin by showing that if $H = \{h_s\}_{s \in S}$ is an ℓ-wise independent family of hash function for sufficiently large ℓ, then for any fixed distribution X with sufficiently large min-Ent(X), for most choices of $s \in S$, $h_s(X)$ is close to the uniform distribution. The interpretation is that for most choices of s, h_s is a good randomness extractor for X. This is formally stated in the next lemma.

Lemma 1. *Let X be an n-bit random variable with* $\text{min-Ent}(X) \geq k$. *Let $H = \{h_s\}_{s \in S}$ be a family of ℓ-wise independent hash functions from n bits to m bits, $\ell \geq 2$. For at least a $1 - 2^{-u}$ fraction of $s \in S$, $h_s(X)$ is ϵ-close to uniform for*

$$u = \frac{\ell}{2}\left(k - m - 2\log(1/\epsilon) - \log \ell + 2\right) - m - 2$$

The proof of uses standard arguments on ℓ-wise independent hash functions (this technique was used in a very related context in [TV02]). We will need the following tail inequality for ℓ-wise independent distributions.

Theorem 5. [BR94] *Let $A_1, \cdots, A_n$ be ℓ-wise independent random variables in the interval $[0,1]$. Let $A = \sum_{i=1}^{n} A_i$ and $\mu = \mathrm{E}(A)$ and $\delta < 1$. Then,*

$$\Pr[|A - \mu| \geq \delta\mu] \leq c_\ell \left(\frac{\ell}{\delta^2 \mu}\right)^{\lfloor \ell/2 \rfloor}$$

Where $c_\ell < 3$ and $c_\ell < 1$ for $\ell \geq 8$.

Proof (of Lemma 1). For $x \in \{0,1\}^n$ let $p_x = \Pr[\mathbf{X} = x]$. We consider the probability space of choosing $s \in_R S$. For every $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$ we define the following random variable:

$$Z_{x,y} = \begin{cases} p_x & h(x) = y \\ 0 & \text{otherwise} \end{cases}$$

We also define $A_{x,y} = Z_{x,y} 2^k$. Recall that for every x, $h_s(x)$ is uniformly distributed, and therefore for every x, y, $\mathrm{E}(Z_{x,y}) = p_x 2^{-m}$. Let $Z_y = \sum_{x \in \{0,1\}^n} Z_{x,y}$ and $A_y = \sum_{x \in \{0,1\}^n} A_{x,y}$. It follows that for every $y \in \{0,1\}^m$, $\mathrm{E}(Z_y) = 2^{-m}$ and thus, $\mathrm{E}(A_y) = 2^{k-m}$. Note that for every x, y, $A_{x,y}$ lies in the interval $[0,1]$ and that for every y, the variables $A_{x,y}$ are ℓ-wise independent. Applying Theorem 5 we obtain that for every y and $\delta < 1$

$$\Pr[|A_y - 2^{k-m}| \geq \delta 2^{k-m}] \leq c_\ell \left(\frac{\ell}{\delta^2 2^{k-m}} \right)^{\ell/2}$$

Substituting Z_y for A_y and choosing $\delta = 2\epsilon$, we get that for every $\epsilon < 1/2$

$$\Pr[|Z_y - 2^{-m}| \geq 2\epsilon 2^{-m}] \leq c_\ell \left(\frac{\ell}{4\epsilon^2 2^{k-m}} \right)^{\ell/2}$$

By a union bound, it follows that with probability $1 - 2^m c_\ell \left(\frac{\ell}{4\epsilon^2 2^{k-m}} \right)^{\ell/2} > 1 - 2^{-u}$ over $s \in_R S$, for all $y \in \{0,1\}^m$ $|Z_y - 2^{-m}| < 2\epsilon 2^{-m}$. We now argue that for such s, $h_s(X)$ is ϵ-close to uniform. Observe that Z_y is the probability that $h_s(X) = y$ (we think of s as fixed, with x chosen according to X). The statistical distance between $h_s(X)$ and the uniform distribution is given by:

$$1/2 \sum_{y \in \{0,1\}^m} |Z_y - 2^{-m}| < 1/2 \sum_{y \in \{0,1\}^m} 2\epsilon 2^{-m} \leq \epsilon$$

■

When applying Lemma 1 with $\ell = 2$, one must set $m < k/2$. This can be avoided as shown by the following lemma, for the special case $\ell = 2$.

Lemma 2. *Let X be an n-bit random variable with* $\text{min-Ent}(X) \geq k$. *Let $H = \{h_s\}_{s \in S}$ be a family of 2-wise independent hash functions from n bits to m. For at least a $1 - 2^{-u}$ fraction of $s \in S$, $h_s(X)$ is ϵ-close to uniform for*

$$u = \frac{k - m}{2} - \log(1/\epsilon) - 1$$

The proof is based on a technique introduced in [ILL89], and will appear in the full version of this paper. The next corollary follows easily, by a union bound.

Corollary 1. *Let $X_1, \cdots, X_{2^t}$ be random variables with values in $\{0,1\}^n$ such that for each $1 \leq i \leq 2^t$,* $\text{min-Ent}(X_i) \geq k$. *Let H and u be as in Lemma 1 (or Lemma 2). For at least a $1 - 2^{t-u}$ fraction of $s \in S$ it holds that for all i, $h_s(X_i)$ is ϵ-close to uniform.*

Theorems 1 and 2 follow by setting $u = log(1/\epsilon)$.