Sustainable Pseudo-random Number Generator

Zhu Huafei, Wee-Siong Ng, and See-Kiong Ng

USPO, I^{2}R, Singapore

Abstract. Barak and Halevi (BH) have proposed an efficient architecture for robust pseudorandom generators that ensure resilience in the presence of attackers with partial knowledge or partial controls of the generators' entropy resources. The BH scheme is constructed from the Barak, Shaltiel and Tromer's randomness extractor and its security is formalized in the simulation-based framework. The BH model however, does not address the scenario where an attacker completely controls the generators' entropy resources with no knowledge of the internal state. Namely, the BH security model does not consider the security of `bad-refresh` conditioned on `compromised = false`. The security of such a case is interesting since if the output of the protocol conditioned on `compromised = false` looks random to the attacker, then the proposed scheme is secure even if the attacker completely controls entropy resources (recall that attackers with partial knowledge or partial controls of the generators' entropy resources in the BH model). The BH scheme is called sustainable if the above mentioned security requirement is guaranteed. This paper studies the sustainability of the BH pseudorandom generator and makes the following two contributions: in the first fold, a new notion which we call sustainable pseudorandom generator which extends the security definition of the BH's robust scheme is introduced and formalized in the simulation paradigm; in the second fold, we show that the BH's robust scheme achieves the sustainability under the joint assumptions that the underlying stateless function G is a cryptographic pseudorandom number generator and the output of the underlying randomness extractor `extract`() is statistically close to the uniform distribution.

Keywords: Provable security, Robust pseudo-random number generator, Sustainable pseudo-random number generator.

1 Introduction

Randomness is essential for security protocols and pseudorandom generators are used to generate random bits from short random seeds [7,8]. A randomness generator, usually is defined over a randomness extractor which in turn is defined over certain mathematical assumptions (e.g., cryptographic hash functions and one-way functions [5,13,4,3,9,12,2,1,14] and the references therein). The reality, however is that procedures for cryptographic systems to obtain random strings are often not well designed [10,6,11].

L.J. Janczewski, H.B. Wolfe, and S. Shenoi (Eds.): SEC 2013, IFIP AICT 405, pp. 139–147, 2013.

Considering a scenario, where a Trusted Platform Module (TPM) is used to collect high-entropy resources so that randomness can be extracted from the generated high-entropy resource (notice that the assumption of high-entropy source is a necessary condition for extracting randomness. This is because one cannot extract m bits from a distribution source with min-entropy less than m). Let X be a random variable describing possible outputs of the TPM in a specified environment. Ideally, we would like the adversary not to be able to influence the distribution of X at all so that the original design for generating high-entropy resource is guaranteed. However, in a realistic setting an adversary may have some control over the environment in which the device operates, and it is possible that changes (e.g., temperature, voltage, frequency, timing, etc.) in this environment affect the distribution of random variable X.

Barak, Shaltiel and Tromer [4] formalized the mathematical mode for the adversary's influence on the source and then proposed an efficient construction of randomness extractors that aim to extract randomness from high entropy resources. They have shown that their randomness extractors work for all resources of sufficiently high-entropy, even the specified resources are correlated. Barak and Halevi [3] then presented formal models and architectures for robust pseudorandom generators (a pseudorandom generator is robust if it is resilient in the presence of attackers with partial knowledge or partial controls of the generators' entropy resources). The Barak and Halevi's (BH) pseudorandom generator consists of the following two algorithms

- A function `next()` that generates output r ($(r, s') \leftarrow$ `next(s)`) and then updates the state s' accordingly; The goal of this component is to ensure that if an attacker does not know the current state s then the output should be random form the point view of the attacker. Typically, `next()` is a deterministic algorithm given an initial state s_0. It is well-known that there exists no single deterministic randomness extractor for all high-entropy resources X and hence the design of `next()` function is a non-trivial task [4,3].
- A function `refresh()` that refreshes the current state s using some additional input x ($s' \leftarrow$ `refresh(s, x)`). The goal of this component is to ensure that if the input x is from a high-entropy resource then the resulting state is unknown to the attacher.

1.1 The Motivation Problem

The security of Barak and Halevi's robust pseudorandom generator is formalized in the simulation-based framework. The security game begins with the system player initializing $s = 0^m$ and compromised =true and then the attacker interacts with the system using the following interfaces:

- `good-refresh`(D) with D a distribution in high entropy source $\mathcal{H}$. The system resets `compromised =false`.
- `bad-refresh`(x) with a bit string x. If `compromised = true` then the system sets $s' =$ `refresh`(s, x) and updates the internal state to s'. Otherwise (if `compromised = false`) it does *nothing*;

- `set-state`(s') with an m-bit string s'. If `compromised = true` then the system returns to the attacker the current internal state s, and if `compromised = false` then it chooses a new random string $s' \leftarrow_R \{0,1\}^m$ and returns it to the attacker.
  Either way, the system also sets `compromised = true` and sets the new internal state to s'.
- `next-bits`(). If `compromised = true` then the system runs $(r, s') = \texttt{next}(s)$, replaces the internal state s by s' and returns to the attacker the m-bit string r. If `compromised = false` then the system chooses a new random string $r \leftarrow_R \{0,1\}^m$ and returns it to the attacker.

Recall that in the ideal world (in the BH model), when an attacker invokes `bad-refresh` conditioned on `compromised = false`, the system does *nothing* while in the real world scenario, even if the attacker invokes `bad-refresh` conditioned on `compromised = false`, the pseudorandom generator scheme will output a refresh statement $s' = \texttt{refresh}(s, x)$. If we consider the security of `bad-refresh`(x) conditioned on `compromised = false`, then there is a security gap between the real-world scenario and the ideal-world scenario in the BH model. Notice that this gap between the ideal world scenario and the real world sceario does not imply that the BH scheme is insecure since the security definition in the BH model does not encompass such a scenario.

1.2 This Work

At first glance, a formalization of sustainable pseudorandom generator is trivial since the state s of the current interface `bad-refresh`(x) conditioned on `compromised = false` is unknown to the attacker in the BH model. We however, aware that to define an output of `bad-refresh`(x) conditioned on `compromised = false`, the following scenarios must be carefully considered

- at least one invocation of `good-refresh`(D) with D in high entropy resource $\mathcal{H}$ has been called before the current invocation of the `bad-refresh`(x) with input x since the initial state of the BH system is 0^m which is publicly known and compromised =true.
- possible many invocations of `next-bits`() have been called since the flag defined in the BH model remains `compromised` = `false` in each `next-bits`() invocation.
- no `set-state`(s') with input s' is invoked between the latest `good-refresh`(D) invocation and the current `bad-refresh`(x) with input x and `compromised = false` invocation;

As a result, the output of `bad-refresh`(x) with input x and `compromised = false` from the point view of the adversary conditioned on the unknown of the current state s should be determined by the transcripts of invocations of the interfaces defined above. Before we define a possible output of `bad-refresh`(x), we would like first to consider the following interesting cases.

- Case 1: Suppose there are total l_1 calls of `good-refresh`(D). The transcript of the l_1 invocations can be expressed in the following form: $s_1 =$ `refresh`(s_0, x_0), $s_2 =$ `refresh`(s_1, x_1), $\cdots$, $s_{l_1} =$ `refresh`(s_{l_1-}, x_{l_1-1}), where s_0 is the initial state and $x_0, \ldots, x_{l_1}$ are selected from the high entropy source $\mathcal{H}$. Notice that $(s_1, \ldots, s_{l_1})$ are kept secret to the attacker.
- Case 2: Suppose there are total l_2 calls of `next-bits`(). We further consider the following two cases:
  Case 2.1: if `compromised = true` then the system runs $(r, s') =$ `next`(s), replaces the internal state s by s' and returns to the attacker the m-bit string (r, s'). More precisely, let s_0 be the initial state that is known to the attacker, then the transcript can be expressed in the following form: $(r_1, s_1) =$ `next-bits`(s_0), $(r_2, s_2) =$ `next-bits`(s_1), $\cdots$, $(r_{l_2}, s_{l_2}) =$ `next-bits`(s_{1_2-1}).
  Case 2.2: if `compromised = false` then the system runs $(r, s') =$ `next`(s), replaces the internal state s by s' and returns r but not s' to the attacker. More precisely, if s_0^* is unknown to the attacker (here we assume that s_0^* is obtained by invoking `good-refresh`(D) interface), then the transcript can be expressed in the following form: $(r_1, s_1) =$ `next-bits`(s_0^*), $(r_2, s_2) =$ `next-bits`(s_1), $\cdots$, $(r_{l_2}, s_{l_2}) =$ `next-bits`(s_{1_2-1}).

Notice that the transcripts useful to the attacker are those generated in Case 2.2 since the transcripts in Case 2.1 can be computed by the attacker itself while the transcripts in Case 1 reveal nothing other than notices of the activated executions. We will define an output of `bad-refresh`(x) conditioned on `compromised = false` a random string. This is because computational indistinguishability is preserved by efficient algorithms. The challenging task now is whether the BH's pseudorandom generator is sustainable? Luckily, we are able to show that the Barak and Halevi's robust pseudorandom generator is sustainable under the joint assumptions that the underlying stateless function G is a cryptographic pseudorandom number generator and the output of the underlying randomness extractor `extract`() is statistically close to the uniform distribution.

Road-Map: The rest of this paper is organized as follows: the notion of sustainable pseudorandom generator is first introduced and formalized in Section 2; We then show that the Barak and Halevi's robust pseudorandom generator is sustainable assuming the existence of cryptographic pseudorandom number generators and the t-resilient extractor and we conclude this work in Section 4.

2 Sustainable Pseudorandom Generators

In this section, we first recall the robust pseudorandom generator due to Barak and Halevi, and then provide a formal definition of sustainable pseudorandom generators

2.1 The Barak and Halevi's Construction

In the high level, Barak-Halevi's framework consists of two functions: `next`(s) that generates the next output and then updates the state accordingly; and `refresh`(s, x) that refreshes the current state s using some additional input x.

- A next function `next` takes as input a sate $s \in \{0,1\}^m$ to generate a pair (r, s'), where r is an l-bit string and s' is an m-bit state. `next` then outputs r and replaces s the internal state by the new state s'.
- A refresh function `refresh` takes as input (s, x) to generate a new state s', where $s \in \{0,1\}^m$ and $x \in \{0,1\}^n$. `refresh` then updates the state with s'.

Definition 1. *Given a collection $H = \{h_\lambda\}_{\lambda \in \Lambda}$ of functions h_λ: $\{0,1\}^n \to \{0,1\}^m$, we consider the probability space of choosing $\lambda \in_R \Lambda$. For every $x \in \{0,1\}^n$, we define the random variable $R_x = h_\lambda(x)$. We say that H is an l-wise independent family of hash functions if:*

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

Lemma 1. *(due to Barak, Shaltiel and Tromer [4]) Let X be a random variable. Let $Pr[X = x]$ be the probability that X assigns to an element x. Let $H_\infty(X) = \log(\frac{1}{max_{x \in X} \Pr[X=x]})$. Let $H = \{h_\lambda\}_{\lambda \in \Lambda}$ be a family of l-wise independent hash functions from n bits to m bits, $l \geq 2$. If $H_\infty(X) \geq k$, then for at least a $1 - 2^{-u}$ fraction of $\lambda \in \Lambda$, $h_\lambda(X)$ is ϵ-close to uniform for $u = \frac{l}{2}(k - m - 2\log(\frac{1}{\epsilon}) - \log(l) + 2) - m - 2)$.*

The function $h_\lambda(X)$ is called randomness extractor. To implement the next function `next` and refresh function `refresh`, Barak and Halevi first invoke the following standard cryptographic pseudorandom generator (`PRG`) [7] and a randomness extractor `extract` [4], where

- `PRG` is a stateless function 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_m is the uniform distribution on $\{0,1\}^m$.
- An extractor is a function `extract`(): $\{0,1\}^{n \geq m} \times \Lambda \to \{0,1\}^m$ for some index set Λ (according to Theorem 1). The output $\texttt{Ext}(x, \lambda)$ is closed to the uniformly distributed, where $x \in \{0,1\}^n$ is the output of the high-entropy source and $\lambda \in \Lambda$.

The BH robust pseudorandom generator

Given an extractor `extract`(): $\{0,1\}^{n \geq m} \to \{0,1\}^m$ and a cryptographic non-robust PRG G: $\{0,1\}^m \to \{0,1\}^{2m}$, where m is a security parameter. By $(r, s') \leftarrow G(s)$, we denote that r is the first m bits in the output of $G(s)$ and s' is the last m bits and by $G'(s) = r$, we denote a function G' that on input $s \in \{0,1\}^m$ outputs only the first m bits of $G(s)$.

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

2.2 Definition of Sustainable Pseudorandom Generator

We follow the BH paradigm that models an attacker on the generator in the real world as an efficient procedure A that has four interfaces to the generator, namely `good-refresh`(), `bad-refresh`(), `set-state`() and `next-bits`(). The ideal-world game proceeds similarly to the real-world game, except that the calls that A makes to its interfaces are handled differently.

The real-World Game. The real world game begins with the system player initializing the internal state of the generator to null, i.e., $s = 0^m$ and then the attacker A interacts with the system using the following interfaces:

- `good-refresh`(D) with D a distribution in $\mathcal{H}$, called high entropy distributions. The system draws $x \leftarrow_R D$, sets $s' =$ `refresh`(s, x) and updates the internal state to s'.
- `bad-refresh`(x) with a bit string x. The system sets $s' =$ `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 $(r, s') \leftarrow$ `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 `PRG` = (`next`, `refresh`), we let $Pr[A(m, H)^{\mathrm{R(PRG)}} = 1]$ 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, H. Here `R(PRG)` stands for the real-world process from above.

The Ideal-World Game. 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 $s = 0^m$ and `compromised =true` and then the attacker interacts with the system using the following interfaces:

- `good-refresh`(D) with D a distribution in $\mathcal{H}$. The system resets `compromised =false`.
- `bad-refresh`(x) with a bit string x. If `compromised = true` then the system sets $s' =$ `refresh`(s, x) and updates the internal state to s'. Otherwise (if `compromised = false`) it outputs a random string s';
- `set-state`(s') with an m-bit string s'. If `compromised = true` then the system returns to the attacker the current internal state s, and if `compromised = false` then it chooses a new random string $s' \leftarrow_R \{0, 1\}^m$ and returns it to the attacker.
  Either way, the system also sets `compromised = true` and sets the new internal state to s'.

– `next-bits`(). If `compromised = true` then the system runs $(r, s') = \texttt{next}(s)$, replaces the internal state s by s' and returns to the attacker the m-bit string r. If `compromised = false` then the system chooses a new random string $r \leftarrow_R \{0,1\}^m$ 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 `PRG` = (`next`, `refresh`), we let I(PRG) denote the ideal process and let $Pr[A(m,H)^{\mathrm{I(PRG)}} = 1]$ denote the probability that A outputs the bit 1 after interacting as above with the system.

Definition 2. *We say that* ***PRG = (next, refresh)*** *is a sustainable pseudo-random generator (with respect to a family $\mathcal{H}$ of distributions) if for every probabilistic polynomial-time attacker algorithm A, the difference*

$$Pr[A(m,H)^{\mathrm{R(PRG)}} = 1] - Pr[A(m,H)^{\mathrm{I(PRG)}} = 1]$$

is negligible in the security parameter l, m and n.

3 The Proof of Security

Theorem 1. *The Barak and Halevi's robust pseudorandom generator is sustainable assuming that the underlying algorithm G is a cryptographic pseudorandom generator and randomness extractor* ***extract*** *with respect to the family $\mathcal{H}$ that is statistically close to the uniform distribution of $\{0,1\}^m$.*

Proof. We consider the following experiments: `Expr.R`, an adversary A interacts with the real system; `Expr.I`, A interacts with the ideal process and `Expr.H`, a hybrid experiment which is defined below:

– `good-refresh`(D) with D a distribution in high entropy resource $\mathcal{H}$. The system draws $d \leftarrow_R \{0,1\}^m$, sets $s' = G'(d \oplus s)$, and updates the internal states to s'. The system resets `compromised =false`.
– `bad-refresh`(x) with a bit string $x \in \{0,1\}^m$. The system sets $s' \leftarrow$ `refresh`($s \oplus$ `extract`(x)) and updates the internal states to s'.
– `set-state`(s') with an m-bit string s'. The system returns to the attacker the current internal state s. The system also sets `compromised = true` and sets the new internal state to s'.
– `next-bits`(s). The system runs $(r, s') =$ `next`(s) and replaces the internal state s by s' and returns to the attacker the m-bit string r.

One checks to see that the only difference between `Expr.R` and `Expr.H` is the definition of the `good-refresh`(D), $D \in \mathcal{H}$. From the construction of randomness extractor [4], we know that the output of randomness extractor is statistically close to U_m. Also note that the output of `good-refresh`(D) with respect to high entropy source $\mathcal{H}$ is statistically close to U_m. As a result, the output of `good-refresh`(D) defined in `Expr.R` and that defined in `Expr.R` are statistically close (statistical closeness is preserved by any function [7]).

Next, we want to show that `Expr.I` and `Expr.H` are computationally close. Suppose that the view of A in `Expr.I` and that in `Expr.H` is distinguishable with non-negligible probability, we construct a challenger B such that given (r^*, s^*), it can distinguish whether it is an output of G for a random $s \leftarrow_R \{0,1\}^m$ or they are chosen at random and independently from $\{0,1\}^m$ with non-negligible property. The challenger B makes use of A as a subroutine and begins by choosing at random an index $i^* \leftarrow \{1, \ldots, q\}$ and setting $s \leftarrow 0^m$ and `compromised = true`. B's ith call of A is answered as follow

- `good-refresh`(D) with D a distribution in $\mathcal{H}$. If $i < i^*$, then the simulator chooses $s' \leftarrow_R \{0,1\}^m$ at random. If $i = i^*$, then the system sets $s' = s^*$, and if $i > i^*$, then the simulator draws $d \leftarrow_R \{0,1\}^m$, sets $s' = G'(d \oplus s)$, where s is the current internal state. Either way, the simulator updates the internal states to s' and resets `compromised =false`.
- `bad-refresh`(x) with a bit string x. If `compromised = true`, the simulator sets $s' \leftarrow$ `refresh`($s \oplus$ `extract`(x)). If `compromised = false` and $i < i^*$, the simulator sets $s' \leftarrow \{0,1\}^m$; and if `compromised = false` and $i = i^*$, then simulator sets $s' = s^*$ and updates the internal states to s'. If `compromised = false` and $i > i^*$, the simulator sets $s' \leftarrow$ `refresh`($s \oplus$ `extract`(x)).
- `set-state`(s') with an m-bit string s'. The simulator returns to the attacker A the current internal state s and sets `compromised = true` and the new internal state to s'.
- `next-bits`(). If `compromised = true` or $i > i^*$ then the simulator set $(r, s') \leftarrow G(s)$. If `compromised = false` and $i < i^*$ then simulator chooses $r, s' \leftarrow \{0,1\}^m$. If `compromised = false` and $i = i^*$, then the simulator sets $r = r^*$ and $s' = s^*$. Either way, the simulator replaces the internal state s by s' and returns to the attacker the m-bit string r.

Let q be a polynomial bounded on the total number of calls made by A to all of its interfaces. Consider the $(q+1)$ experiments $H^{(i)}$, $i = 0, 1, \ldots, q$, 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 $i > i^*$. We claim that $H^{(q)} =$ `Expr.I` and $H^{(0)} =$ `Expr.R`. Let $\Pr[\text{Dist}(H^{(0)}) = 1] = \delta_0$ and $\Pr[\text{Dist}(H^{(q)}) = 1] = \delta_q$

$$
\begin{aligned}
|\delta_0 - \delta_q| &= |\sum_{i=1}^{q} \Pr[\text{Dist}(H^{(i)}) = 1] - \Pr[\text{Dist}(H^{(i-1)}) = 1]| \\
&\leq \sum_{i=1}^{q} |\Pr[\text{Dist}(H^{(i)}) = 1] - \Pr[\text{Dist}(H^{(i-1)}) = 1]| \\
&\leq q\varepsilon
\end{aligned}
$$

This means that if the view of `Expr.I` and the view of `Expr.R` are distinguishable with non-negligible probability, then we are able to distinguish whether (r^*, s^*) is an output of G for a random $s \leftarrow_R \{0,1\}^m$ or they are chosen at random and independently from $\{0,1\}^m$ with non-negligible advantage.

4 Conclusion

In this paper, we have introduced and formalized the notion of sustainable pseudorandom generator which aims to fill the security gap between the ideal world and the real world in the BH robust pseudorandom generator. We have shown that the Barak and Halevi's construction is sustainable assuming that the underlying algorithm G is a cryptographic pseudorandom number generator and the output of the underlying randomness extractor is statistically close to the uniform distribution U_m.

References

1. De, A., Watson, T.: Extractors and Lower Bounds for Locally Samplable Sources. TOCT 4(1), 3 (2012)
2. Boldyreva, A., Kumar, V.: A New pseudorandom Generator from Collision-Resistant Hash Functions. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 187–202. Springer, Heidelberg (2012)
3. Barak, B., Halevi, S.: A model and architecture for pseudorandom generation with applications to /dev/random. In: ACM Conference on Computer and Communications Security, pp. 203–212 (2005)
4. Barak, B., Shaltiel, R., Tromer, E.: True Random Number Generators Secure in a Changing Environment. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 166–180. Springer, Heidelberg (2003)
5. Blum, M., Micali, S.: How to Generate Cryptographically Strong Sequences of Pseudo Random Bits. In: FOCS 1982, pp. 112–117 (1982)
6. Dorrendorf, L., Gutterman, Z., Pinkas, B.: Cryptanalysis of the windows random number generator. In: ACM Conference on Computer and Communications Security, pp. 476–485 (2007)
7. Goldreich, O.: Foundation of Cryptography, vol. I. Cambridge University Press (2001)
8. Goldreich, O.: Foundation of Cryptography, vol. II. Cambridge University Press (2004)
9. Goldreich, O., Izsak, R.: Monotone Circuits: One-Way Functions versus pseudorandom Generators. Electronic Colloquium on Computational Complexity (ECCC) 18, 121 (2011)
10. Gutterman, Z., Pinkas, B., Reinman, T.: Analysis of the Linux Random Number Generator. In: S&P 2006, pp. 371–385 (2006)
11. Goldberg, I., Wagner, D.: Randomness and the Netscape Browser. Dr. Dobb's Journal, 66–70 (1996)
12. Kamp, J., Rao, A., Vadhan, S.P., Zuckerman, D.: Deterministic extractors for small-space sources. J. Comput. Syst. Sci. 77(1), 191–220 (2011)
13. Yao, A.C.-C.: Theory and Applications of Trapdoor Functions (Extended Abstract). In: FOCS 1982, pp. 80–91 (1982)
14. Vadhan, S.P., Zheng, C.J.: Characterizing pseudoentropy and simplifying pseudorandom generator constructions. In: STOC 2012, pp. 817–836 (2012)