Distributed Pseudo-Random Number Generation and its application to Cloud Database

Jiageng Chen, Atsuko Miyaji, and Chunhua Su

School of Information Science,
Japan Advanced Institute of Science and Technology, Japan.
{$jg-chen, miyaji, su$}@jaist.ac.jp

Abstract. Cloud database is now a rapidly growing trend in cloud computing market recently. It enables the clients run their computation on out-sourcing databases or access to some distributed database service on the cloud. At the same time, the security and privacy concerns is major challenge for cloud database to continue growing. To enhance the security and privacy of the cloud database technology, the pseudo-random number generation (PRNG) plays an important roles in data encryptions and privacy-preserving data processing as solutions. In this paper, we focus on the security and privacy risks in cloud database and provide a solution for the clients who want to generate the pseudo-random number collaboratively in a distributed way which can be reasonably secure, fast and low cost to meet requirement of cloud database. We provide two solutions in this paper, the first one is a construction of distributed PRNG which is faster than the traditional Linux PRNG. The second one is a protocol for users to execute the random data perturbation collaboratively before uploading the data to the cloud database.

Keywords: cloud database, pseudo-random number generators, distributed computation, data randomization

1 Introduction

In the so-called "big data" era, huge volumes of data are being created from the organizations procedure, business activities, social network and scientific research. Databases are ubiquitous and of immense importance and the cloud database technology offers many benefits such as data storage and outsource computing to meet the new technological requirements. Many cloud database service and computation are in the distributed environment, as an important security primitive, pseudo-random number generator play an extremly important role in such cloud based data service. In this paper, we propose a framework for pseudo-random number generator (PRNG) which is used in distributed cloud database, our proposal is based on the collection of high entropy from operation system such as Linux.

In Linux PRNG, there are two devices dev/random and dev/urandom. dev/random is nearly a true random number generator consists of a physical non deterministic phenomenon produces a raw binary sequence and a deterministic function, compress this sequence in order to reduce statistical weakness. But these procedures to produce the nearly true random number sequences from dev/random are expensive and low-speed for practical cloud database application. So usually for practical usage, we use pseudo-random number generators which are deterministic random bit generators such as Linux dev/urandom, Linux dev/urandom use algorithms for generating a sequence of numbers that approximates the properties of true random number but with lower security bound. Our research purpose is to make the robust and secure dev/random which is more secure and robust to be faster to meet the need of cloud database service.

1.1 Related works

Many random number generators exists (e.g., [15, 18, 23, 1, 20, 21]. Shamir was first to provide SPRNG [23] while Blum-Blum-Shub [1] and many other PRNGs followed. A high-quality source of randomness must be used to design a high-quality true random data generator for cryptographic purposes. In a typical environment of general purpose computer systems, some good sources of randomness may exist in almost any user input - the exact timing of keystrokes and the exact movements of mouse are well known. Some other possible sources are for example microphone (if unplugged then A/D convertor yields electronic noise [9]), video camera (focused ideally on some kind of chaotic source as lava lamp [16]), or fluctuations in hard disk access time [6].

Following the unsuitability of the so called statistical PRNGs for cryptographical purposes, special PRNGs, intended for cryptography uses, were developed. The most related works to ours are Linux PRNG. The first security analysis of Linux PRNGs was given in 2006 by Gutterman *et. al* [10], based on kernel version 2.6.10 released in 2004. In 2012, Lacharme *et. al* [19] gave a detailed analyze the PRNG architecture in the Linux system and provide its first accurate mathematical description and a precise analysis of the building blocks, including entropy estimation and extraction.

1.2 Problem Definition and Our Contributions

There are two common deployment models of cloud database: users can run databases on the cloud independently, using a virtual machine image, or they can purchase access to a database service, maintained by a cloud database provider such as Distributed database as a service (DBaaS). However, cloud database adoption may be hampered by concerns about security, privacy, and proprietary issues, such distributed DBaaS are vulnerable to threats such as unauthorized access and malicious adversaries who want to compromise the privacy of the data storage. Our protocol is constructed based on Linux kernel, the internal state of the Linux PRNG is composed of three pools, namely the input pool, the blocking pool for dev/random output and the nonblocking pool for dev/urandom

output, according to the source code. We assume that there are many severs who provide outsourcing distributed database services and need to generate pseudo-random number for encryption or data random perturbation. If the blocking pool cannot accumulate enough entropy, the PRNG output will be blocked. However, the Linux OS dev/random is extremely suitable for use when very high quality randomness is desired (for example, for key generation, one-time pads and sensitive data randomization), but it will only return a maximum of the number of bits of randomness contained in the entropy pool. The major problem we focus in this paper is to construct a fast dev/random in the distributed environment to achieve a higher speed.

- For the cryptographic purposes, the distributed clients and the cloud servers may need to generate encryption or decryption keys to secure their communication or create some fresh nonce or to execute the protocols for authentication. In this case, pseudo-random numbers are necessary for both key generations, encryption authentication.
- For data privacy purpose, the clients who purchase the services for the cloud database may store their database on the cloud servers. In order to aggregate information that contains personal data without involving a trusted aggregator, two important constraints must be fulfilled: 1) privacy of individuals or data subjects whose data are being collected, and 2) security of data contributors who need to protect their data from other contributors as well as the untrusted aggregation.

For an OS-based pseudo-random generator, Linux PRNG is a good candidate for the distributed environment. Because it is an open-source OS and it plays a huge role in virtualized cloud operations including the DBaaS. The theory of computational pseudo-randomness discussed in our paper emerged from cryptography, where researchers sought a definition that would ensure that using pseudo-random bits instead of truly random bits would retain security against all computationally feasible attacks.

Our Contributions In this paper, we propose a framework for pseudo-random number generators under the distributed environment.

- We clarify the necessary conditions for implementing secure and fast PRNGs for the distributed cloud database.
- We propose a protocol based on Linux PRNG for the distributed pseudo-random number generation which is faster than stand-alone Linux PRNG. We let all parties execute the collection of entropy for distributed random source and then mix them securely to form a local random pool for the pseudo-random number generation. The second one is to apply Barak-Shaltiel-Tromer randomness extractor randomness exactor to generate the pseudo-random number with the same probability distribution for the data aggregation in cloud database.
- We also provide the security proof and show that the security of our proposals holds as long as the adversary has limited influence on the high-entropy source.

The rest of the paper is constructed as follows: We outline preliminaries for pseudo-random number generator in Section 2. The constructions of our schemes are in Section 3 and Section 4, respectively. In Section 5, we provide the security proofs for our proposed distributed PRNGs and our experimental analysis. We draw the conclusions in Section 6.

2 Preliminaries

In this section, we give a brief descriptions about the building blocks used in our schemes and the security definition.

2.1 Building Blocks

Linux PRNG. The Linux PRNG is part of the Linux kernel since 1994. The original version was written by Tsfo [24], and later modified by Mackall [22]. The PRNG is implemented in C with more than about 1700 lines code in a single source file, drivers/char/random.c. There are many build-in function which we can use to construct our distributed PRNG.

Barak-Halevi Model. Let us briefly recall construction of PRNG with input due to Barak and Halevi [3]. This model (which we call BH model) involves a randomness extraction function: $Extract : \{0,1\}^p \rightarrow \{0,1\}^n$ and a standard deterministic PNRG $G; \{0,1\}^n \rightarrow \{0,1\}^{n+l}$ In the Barak-Halevi's framework, two functions are defined in the pseudo-random number generator: function $\mathsf{next}(s)$ that generates the next output and then updates the state accordingly and a function $\mathsf{refresh}(s, x)$ that refreshes the current state s using some additional input x.

Twisted Generalized Feedback Shift Register (TGFSR)[20]. It is a improved version of Generalized Feedback Shift Register (GFSR) which can be used to run w Linear Feed Back Registers (LFSR) in parallel, where w is the size of the machine word and its cycle length $2^p - 1$ with a memory of p words. TGFSR achieves a period of $2^{wp} - 1$ and removes the dependence of a initialized sequence in GFSR, without the necessary of polynomial being a trinomial.

Verifiable Secret Sharing. The VSS protocol has a two-phase structure: In a primary phase, the dealer D distributes a secret s, while in a second, later phase, the players cooperate in order to retrieve it. More specifically, the structure is as follows:

- Sharing phase: The dealer initially holds secret $s \in K$ where K is a finite field of sufficient size; and each player P_i finally holds some private information v_i.
- Reconstruction phase: In this phase, each player P_i reveals (some of) his private information v_i. Then, on the revealed information v_i' (a dishonest player may reveal $v_i' \neq v_i$), a reconstruction function is applied in order to compute the secret, $s = Rec(v_1', ..., v_n')$

2.2 Security Definitions and Model

A deterministic function $G : \{0,1\}^d \rightarrow \{0,1\}^m$ is a (t, ϵ) pseudo-random generator (G) if $d < m$, $G(U_d)$ and U_m are (t, ϵ) indistinguishable. Information disclosure in G refers to the leaking of the internal state, or seed value, of a PRNG. Leaks of this kind can make predicting future output from the PRNG in use much easier. Here in this paper. we follow the formal security model for PRNGs with input was proposed in 2005 by Barak and Halevi (BH model) [3] and its extension by Dodis *et al.* [8].There is a proof that the security definition imply the following security notions in [8].

- Resilience: The internal state and future output of PRNG must not able to predict even if an adversary can influence or attain the entropy source used to initialize or refresh the internal state of the PRNG;
- Forward security and backward security: an adversary must not be able to predict past and future outputs even if he can compromise the internal state of the PRNG.

Our security model is based on Dodis *et al. [8]*'s modified BH model, the adversary $\mathcal{A}$ can access several oracle calls as follows:

- D-refresh. This is the key procedure where the distribution sampler D is run, and where its output I is used to refresh the current state ST. Additionally, one adds the amount of fresh entropy to the entropy counter c, and resets the corrupt flag to false when c crosses the threshold γ.
- get-state and set-state. These procedures provide $\mathcal{A}$ with the ability to either learn the current state ST, or set it to any value ST^*. In either case c is reset to 0 and corrupt is set to true. We denote by q_S the total number of calls to get-state and set-state.
- next-ror and get-next. These procedures provide $\mathcal{A}$ with either the real-or-random challenge (provided that corrupt = false) or the true PRNG output. As a small subtlety, a gprematureh call to get-next before corrupt = false resets the counter c to 0, since then $\mathcal{A}$ might learn something non-trivial about the (low-entropy) state ST in this case. We denote by q_R the total number of calls to next-ror and get-next.

This model involves an internal state that is refreshed with a (potentially biased) external random source, and a cryptographic function that outputs random numbers from the continually internal state. The game continues in this fashion until the attacker decides to halt with some output in $\{0,1\}$. For a particular construction $G = (\mathsf{setup}, \mathsf{next}, \mathsf{refresh})$, we let $Pr[A(m,H)^{\mathrm{I(G)}} = 1]$ denote the probability that adversary $\mathcal{A}$ outputs the bit 1 after interacting as above with the system. Here I(G) stands for the ideal random process and note that we only use G in this game to answer queries that are made while the compromised flag is set to true.

Definition 1. *We say that G = (setup, next, refresh) is a robust pseudo-random generator (with respect to a family $\mathcal{H}$ of distributions) if for every probabilistic polynomial-time attacker algorithm $\mathcal{A}$, the difference*

$$Pr[A(m,H)^{\mathrm{G}} = 1] - Pr[A(m,H)^{\mathrm{I(G)}} = 1] < \epsilon$$

in some security parameter as follows:

- *G with input has $(t, qD, \gamma^*, \epsilon)$-recovering security if for any adversary $\mathcal{A}$ and legitimate sampler D, both running in time t, the advantage of recovering the internal state with parameters q_D is at most ϵ.*
- *G with input has (t, ϵ)-preserving security if the advantage of any adversary $\mathcal{A}$ running in time t of distinguishable G output and internal state from true random sample is at most $a\epsilon$.*

3 Our Proposal on Distributed PRNG

The PRNG used by the cloud server relies on external entropy sources. Entropy samples are collected from system events inside the kernel, asynchronously and independently from output generation. These inputs are then accumulated into the input pool. Beyond the difficulty of collecting truly random data from various randomness sources, the problem of insufficient amount of truly random data which can be effectively solved by using pseudo-random data is also important. Our protocol overcomes this problem by sharing the collecting the entropy in cloud computing environment.

1. We apply a hash function or symmetric key encryption scheme to protect the vulnerable PRNG outputs. If a PRNG is suspected to be vulnerable to direct cryptanalytic attack, then outputs from the PRNG should be preprocessed with a cryptographic hash function.
2. Occasionally generate a new starting PRNG state, a whole new PRNG state should occasionally be generated from the current PRNG. This will ensure that any PRNG can fully reseed itself, given enough time and input entropy. The best way to resist all the state-compromise extension attacks is simply never to have the PRNG's state compromised.

3.1 Distributed Pseudo-random Number Generator

A nice PRNG should always have a component for harvesting entropy. Even if entropy is only used to seed a PRNG, the infrastructures using PRNG should still harvest their own entropy, because experience shows that pawning the responsibility for entropy harvesting onto clients leads to a large number of systems with inadequately seeded PRNGs. Entropy gathering should be a separate component from the PRNG. This component is responsible for producing outputs that are believed to be truly random. The following work reviewed is due to Gutterman, Pinkas and Reinman [10].

Our work focus on how to overcome the security problems in existing PRNG based on Linux. Furthermore, we apply the Lacharme's linear corrector [17] to implement the entropy addition and update to get more high entropy compared to existing Linx-based PRNG [10].

We assume that there are k distributed users or cloud servers online to generate the pseudo-random number. Let $G_i : \{0,1\}^m \to \{0,1\}^{n+l}$ be a distributed pseudo-random generator, where $1 \leq i \leq k$, and a ensemble of external input I. We then model our PRNG construction as follows:

- Initial phase: It uses a function setup() to generate the $seed = (s, s') \leftarrow \{0,1\}^{n+l}$ at first.
- State refresh phase: Given $seed = (s, s')$ as input, the refresh algorithm refresh(ST, I) outputs a next internal state ST'
- Random bits output phase: The generator G_i outputs a random string R and a new state ST'.

The Protocol for Distributed Pseudo-random Number Generation

1. Each party translates system events into bits which represent the underlying entropy and then share out their entropy to other parties.
2. Each party also collecting entropy for other other party. Because the system cannot consume more entropy than it has collected, and once the entropy counter for the input pool has reached its maximum value, the party starts ignoring any incoming events in order to save CPU resources. Thus, it collects no more entropy.
3. Each party run the setup() to get the initial seeds as input of PRNG G, after that each party adds these bits to the generator pools using add_input_randomness() function in Linux PRNG.
4. Each party use function refresh() to extract entropy and update the entropy pool. If the accumulated entropy from both internal events and external events of other parties can pass the test of entropy estimator, send the bits as input of function next().
5. When bits are read from the generator, each party uses function next() to generates the output of the generator and the feedback which is entered back into the pool.
6. Each party runs its internal G, let the random data generation be done in blocks of 10 output bytes. For each output block, 20 bytes, produced during the process, are injected and mixed back into the source pool to update it.

Fig. 1. The Distributed PRNG

3.2 The details of our Protocol

There are four asynchronous procedures: the initialization, the entropy accumulation, pool updating with entropy addition and the random number output. We provide some details of our construction in the following paragraphs.

Initialization Operating system start-up includes a sequence of routine actions. This sequence, including initializing the PRNG with constant OS parameters and time-of-day, can easily be predicted by an adversary. If no special actions are taken, the PRNG state will include very low entropy. The time of day is given as seconds and micro-seconds, each is 32-bits. In reality this has very limited entropy as one can find computer uptime within an accuracy of a minute, which leads to a brute-force search of $60_{seconds} \times 106_{microseconds} < 2^{26}$ which is feasible according the [11]. Even if the adversary cannot get the system uptime, he can check the last modification time of files that are created or modified during the system start-up, and know the uptime in an accuracy of minutes.

To solve this problem, PRNG simulate continuity along shut-downs and start-ups. This is done by skipping system boots. A random-seed is saved at shut-down and is written to the pools at start-up. A script that is activated during system start-ups and shut-downs uses the read and write capabilities of /dev/urandom interface to perform this maintenance.

The script saves 512 bytes of randomness between boots in a file. During shut-down it reads 512 bytes from /dev/random and writes them to the file, and during start-up these bits are written back to /dev/random. Writing to /dev/random modifies the primary pool and not the random pool, as one could expect. The secondary and the random pool get their entropy from the primary pool, so the script operation actually affects all three pools.

The author of [24] instructs Linux distribution developers to add the access control of initial seed in order to ensure unpredictability at system start-ups. This implies that the security of the PRNG is not completely stand-alone but dependent on an external component which can be predictable in a certain Linux distribution.

2. Collecting and Sharing Entropy Each party collects entropy from events originating from the keyboard, mouse, disk and interrupts on each client's local computer while collecting the event entropy from other parties. When such an event occurs, four 32-bit words are used as input to the entropy pools: For each entropy event fed into the Linux PRNG, three 32-bit values are considered: the num value, which is specific to the type of event 2, the current CPU cycle count and the jiffies count at the time the event is mixed into the pool. Here, we can use three functions for Linux PRNG: add_disk_randomness(), add_input_randomness() and add_interrupt_randomness().

The sequence from the three function represent the $jiffies$ counts (the time between two ticks of the system timer interrupt) of the events, and is thus an increasing sequence. Since the estimation of the entropy should not depend on the time elapsed since the system was booted (beginning of the jiffies count),

only the sequence of time differences are considered. A built-in estimator *Ent* is used to give an estimation of the entropy of the input data used to refresh the state ST. It is implemented in function add_timer_randomness which is used to refresh the input pool.

3. Entropy Addition and Pool Updating LINUX PRNG uses an internal mixing function which is implemented in the built-in function mix_pool_bytes. It is used in two contexts, once to refresh the internal state with new input and secondly to transfer data between the input pool and the output pools., used to refresh the internal state with new input and to transfer data between the pools. The design of the mixing function relies on a Twisted Generalized Feedback Shift Register (TGFSR) as defined in Section 2. In the entropy pools, we add Lacharme's linear corrector with mixing function to update the pool, it is a deterministic function to compress random sequence in order to reduce statistical weakness [17]. Let C the $[255, 21, 111]$ BCH code, D the $[256, 234, 6]$ dual code of C, with generator polynomial

$$H(X) = X^{21} + X^{19} + X^{14} + X^{10} + X^7 + X^2 + 1 \tag{1}$$

The input 255-tuple $(m^1, ..., m^{255})$ is coded by a binary polynomial $m(X) = \sum^2 55_{i=1} m_i X^i$. Therefore the function f mapping F_2^{255} to F_2^{21}, defined by $m(X) \mapsto m(X) \mod H(X)$ is a (255, 21, 110)-resilient function. This polynomial reduction is implemented by a shift register of length 21 with only seven xor gates.In this case, with an important input bias $e/2 = 0.25$, it give an output bias of: $\forall y \in F_2^{21} | P(f(X) = y) - 2^{-21}| \leq 2^{-111}$. Therefore, the minimal entropy of the output is very close to 21.

We can use a general constructions of good post-processing functions. We have shown that linear correcting codes and resilient functions provide many correctors achieving good bias reduction with variable input sizes. Linear feedback shift register are suitable for an hardware implementation where the chip area is limited.

If input pool does not contain enough entropy. Otherwise, estimated entropy of the input pool is increased with new input from external event. Entropy estimation of the output pool is decreased on generation. Data is transferred from the input pool to the output pools if they require entropy. When the pools do not contain enough entropy, no output can be generated with /dev/random.

4. Random Bits output Entropy estimations of the participating pools are updated according to the transferred entropy amount. Extracting entropy from a pool involves hashing the extracted bits, modifying the pool inner-state ST and decrementing the entropy amount estimation by the number of extracted bits. Such tasks are executed by next() function in G. It extracts 80 random bytes from the secondary pool one time.It uses SHA-1 and entropy-addition operations before actually outputting entropy in order to avoid backtracking attacks. In addition it uses folding to blur recognizable patterns from 160 bits SHA-1 output into 80 bits.

Once mixed with the pool content, the 5 words computed in the first step are used as an initial value or chaining value when hashing another 16 words from the pool. These 16 words overlap with the last word changed by the feedback data. In the case of an output pool (pool length = 32 words), they also overlap with the first 3 changed words. The 20 bytes of output from this second hash are folded in half to compute 11 the 10 bytes to be extracted: if $w\,[m...n]$ denotes the bits $m, ..., n$ of the word w, the folding operation of the five words w_0, w_1, w_2, w_3, w_4 is done by $w_0 \oplus w_3, w_1 \oplus w_4, w_2\,[0...15] \oplus w_2\,[16...31]$. Finally, the estimated entropy counter of the affected pool is decremented by the number of generated bytes.

4 Application to Distributed Data Random Perturbation

Data random perturbation is a technology when preserve the data privacy by adding the random noise to the original data, in recent years, it has been reviewed and the such as differential privacy is the state-of-the-art privacy notion [7] that gives a strong and provable privacy guarantee for aggregated data. The basic idea is partial random noise is generated by all participants, which draw random variables from Gamma or Gaussian distributions, such that the aggregated noise follows Laplace distribution to satisfy differential privacy.

Here in this section, we propose a application of our distributed PRNG. Combined with randomness extractor We assume that 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. In the Barak-Shaltiel-Tromer model, they assumed that the adversary can control at most t boolean properties of the environment, and can thus create at most 2^t different environments.

Definition 2. *(Barak-Shaltiel-Tromer randomness extractor [2]) A function $\mathcal{E}$: $\{0,1\}^n \times \{0,1\}^t \to \{0,1\}^m$ is a (k, ϵ)-extractor if for every random variable X of min-entropy at least k it holds that $\mathcal{E}(X, U_t)$ is ϵ-close to the uniform distribution over $\{0,1\}^m$.*

Definition 3. *(The security definition of Barak-Shaltiel-Tromer randomness extractor [2])*

- *An adversary chooses 2^t distributions $D_1, \ldots, D_{2^t}$ over $\{0,1\}^n$, such that $H_\infty(D_i) > k$ for all $i = 1 \cdots, 2^t$.*
- *A public parameter π is chosen at random and independently of the choices of D_i.*
- *The adversary is given π, and selects $i \in \{1, ..., 2^t\}$.*
- *The user computes $\mathcal{E}^\pi(X)$, where X is drawn from D_i.*

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

We can apply this random extractor to a distributed environment with k participated database owners $P_i, i = 1, ..., k$ by using the Verifiable Secret Sharing scheme [5] which allows any party distributes his shares of a secret, which can be verified for consistency. The Gaussian noise can be generated and all the participants cooperatively verify that the shared values are legitimate. Finally, each party site $P_i, i = 1, ..., n$ can cooperatively reconstruct the original data density by using the reconstruction technique of the verifiable secret sharing scheme. In our proposal, we need that all parties must generate the noise from the same probability distribution. y_i and b_i can be generated interactively among all parties. The protocol is shown in Fig.2.

Protocol for distributed random noise generation for cloud database service

1. Before upload data to cloud database, the client i will generate random bits string using the G we proposed in last section and get $a_{1,i}, ..., a_{k,i}$.
2. Every client collaborative executing the coin tossing protocol some random bits $b_1, ..., b_n$ with a pre-determined distribution and share out those bits via verifiable secret sharing.
3. The client i can apply a randomness extractor $Ext()$ with the input $\pi_i(b_1, ..., b_n)$ where π_i is a random permutation and get the random bits $c_{1,i}, ..., c_{k,i}$
4. Then client i computes $a_{1,i} \oplus c_{k,i}, ..., a_{1,i} \oplus c_{k,i}$ and then converts these random bits $GF(2)$ to random noise on $GF(q)$. The sequences of random noise is serially uncorrelated and the output has good statistical characteristics due to the randomness extractor.

Fig. 2. The Distributed Random Noise Generation

Each party i shares a random bit by sharing out a value $b_i \in \{0, 1\}_{GF(q)}$, using a non-malleable verifiable secret sharing scheme, where q is sufficiently large, and engages in a simple protocol to prove that the shared value is indeed in the specified set. And then suppose for a moment that we have a public source of unbiased bits, $c_1, c_2, ..., c_n$. By XORing together the corresponding b's and c's, we can transform the low quality bits b_i (in shares) into high-quality bits $b_i \oplus c_i$ in shares. Finally, each participant party sums her shares to get a share of the random noise.

The principal costs are the multiplications for verifying random noise in $\{0, 1\}_{GF(q)}$ and the executions of verifiable secret sharing. Note that all the verifications of random noise parameters are performed simultaneously, so the messages from the different executions can be bundled together. The same is true for the verifications in the VSS. The total cost of the scheme is $\Theta(n)$ multiplications and additions in shares, which can be all done in a constant number of rounds.

5 Security Proofs and Experimental Analysis

In this section, we provide security proofs and the experimental analysis.

5.1 Security Proof of the distributed PRNG

We show the security of our scheme in Theorem 1 and Theorem 2 as follows:

Theorem 1. *Our PRNG has t', q_D, ϵ-recovering security.*

Proof. The adversary $\mathcal{A}$ compromised a state and get the value ST_0 of the G, let's consider the game as follows:

- The challenger choose a seed $seed \leftarrow setup()$, after that he sample the D and get some assemble $\sigma_k, I_k, \gamma_k, z_k)$, where $k = 1, ..., q_D$. Here γ is some fresh entropy estimation of I, z is the leakage about I given to the $\mathcal{A}$.
- The adversay get $\gamma_1, ..., \gamma_{q_D}$ and $z_1, ..., z_{q_D}$, after that he launch an attack against $q_D + 1$ step computation of G, he can call $\mathsf{D\text{-}fresh}$ along with other oracle
- The challenger sequentially computes $ST_j = \mathsf{refresh}(ST_{j-1}, I_j, seed)$ for $j = 1, ..., d$. If $b = 0$, $\mathcal{A}$ is given $(ST^*, R) = \mathsf{next}(ST_d)$ and if $b = 1$, $\mathcal{A}$ is given $(ST^*, R) \leftarrow \{0,1\}^{n+l}$.
- The adversary $\mathcal{A}$ output a bit b^*.

Adversary can query the oracles in security definition and try to distinguish the internal state from the random sample. Let Game 0 be the original recovering security game above: the game outputs a bit which is set to 1 iff the $\mathcal{A}$ guesses the challenge bit $b^* = b$. We define Game1 where, during the challengerfs computation of $(ST^*, R) \leftarrow \mathsf{next}(S_d)$ for the challenge bit $b = 0$, it chooses $U \leftarrow \{0,1\}^m$ uniformly. We can know that $\Pr[(Game0) = 1] - \Pr[(Game1) = 1] \leq \epsilon$ according the argument in [8].

Theorem 2. *Our PRNG has t, ϵ-preserving security.*

Proof. Intuitively, it says that if the state S_0 starts uniformly random and uncompromised, and then is refreshed with arbitrary (adversarial) samples. Here in this paper, we adapt the security notions which is simplified by Dodis [8] based on BH model we mentioned above. $I_1, ..., I_d$ resulting in some final state S_d, then the output $(S^*, R) \leftarrow \mathsf{next}(S_d)$ looks indistinguishable from uniform. Formally, we consider the following security game with an adversary $\mathcal{A}$ who try to compromise the PRNG. We consider the game as follows:

- The challenger chooses an initial state $S_0 \leftarrow \{0,1\}^n$, a seed $seed \leftarrow \mathsf{setup}$, and a bit $b \leftarrow \{0,1\}$ uniformly at random.
- $\mathcal{A}$ gets seed and specifies arbitrarily long sequence of values $I_1, ..., I_d$ with $I_j \in \{0,1\}^n$ for all $j \in [d]$.
- The challenger sequentially computes $ST_j = \mathsf{refresh}(ST_{j-1}, I_j, seed)$ for $j = 1, ..., d$. If $b = 0$, $\mathcal{A}$ is given $(ST^*, R) = \mathsf{next}(ST_d)$ and if $b = 1$, $\mathcal{A}$ is given $(ST^*, R) \leftarrow \{0,1\}^{n+l}$.

– A outputs a bit b^*.

Without loss of generality, we will assume that all compromised next queries that the $\mathcal{A}$ makes are to $get - next$. Let Game 0 be the original preserving security game: the game outputs a bit which is set to 1 iff the attacker guesses the challenge bit $b^* = b$. If the initial state is $ST_0 \leftarrow \{0,1\}^n$, the seed is $seed = (X, X')$, and the adversarial samples are $I_d, ..., I_0$ (indexed in reverse order where I_d is the earliest sample) then the refreshed state that incorporates these samples will be $S_d := S_0 \cdot X_d + P_j = I_j \cdot X_j$. As long as $X = 0$, the value ST_d is uniformly random (over the choice of S_0). We consider a modified Game 1, where the challenger simply chooses $ST_d \leftarrow \{0,1\}^n$. We can use using the hybrid argument and get the advantage of $\mathcal{A}$ is $|\Pr[b] - \frac{1}{2}| < \epsilon$.

5.2 Security Analysis of application of PRNG in random data perturbation

In order to determine the effect of a perturbation method, it is necessary to consider the security provided by that method. If two data matrics X and X' differ in a single row, the statistical difference between X and X' is 1/n. Let X be a random variable and $\Pr[X = x]$ be the probability that X assigns to an element x. Let $H_\infty(X) = \log(\frac{1}{max_{x \in X} \Pr[X=x]})$. By definition, it is easy to verify that the following claims:

– If $\max_{x \in X} \Pr[X = x] \leq 2^{-k}$ if and only if $\mathrm{H}_\infty(X) \geq k$;
– If $\max_{x \in X} \Pr[X = x] \geq 2^{-k}$ if and only if $\mathrm{H}_\infty(X) \leq k$.

To design randomness extractor $\mathcal{E}$: $\{0,1\}^n \rightarrow \{0,1\}^m$, we need to consider its input and mathematical structure. It is well-known result that one cannot extract m bits from a distribution X with $H_\infty(X) \leq m - 1$. $H_\infty(X) \leq m - 1$ implies $Pr[X = x] \geq 2^{-(m-1)}$. For any candidate extractor function $\mathcal{E}$: $\{0,1\}^n \rightarrow \{0,1\}^m$, we know that $Pr[y = \mathcal{E}(x)] \geq 2^{-(m-1)}$. It follows that $\mathcal{E}(x)$ is far from being uniformly distributed. Another well-known result is that there exists no single deterministic randomness extractor for all high-entropy sources X. Consider the goal of designing an extractor for all distributions X with $H_\infty(X)$ $\leq n-1$. One can show that there exists a design for function $\mathcal{E}$: $\{0,1\}^n \rightarrow \{0,1\}$. For an arbitrary adversary $\mathcal{A}$, there are two statistically similar data matrix, only differ in on row, after the linear transformation, he can not indistinguish between the transcript $T(X)$ and $T(X')$. Because $T(X)/T(X')$ is at most $e^{-\frac{X-X'}{\sigma}}$, where σ is . Using the law of conditional probability, and writing t_i for the indices of t, $\frac{\Pr(T(X)=t)}{\Pr(X'+Y=t)} \in exp(\pm\frac{|X-X'|}{\sigma})$.

5.3 Experimental Analysis

Our experiment is executed on Note PCs with 2.6GHz, the OS is 32-bit Ubuntu 13.10. We collect the entropy from three other PC and generate the pseudo-random from 100 bytes to 1000 bytes. We did not use linux kernel APIs in

linux/net.h and /linux/netpoll.h to send UDP packets, but to collect three PC's entropy and form them into a file which is used for the fourth experiment PC. So we only calculate the computation time as shown in Fig. 3.

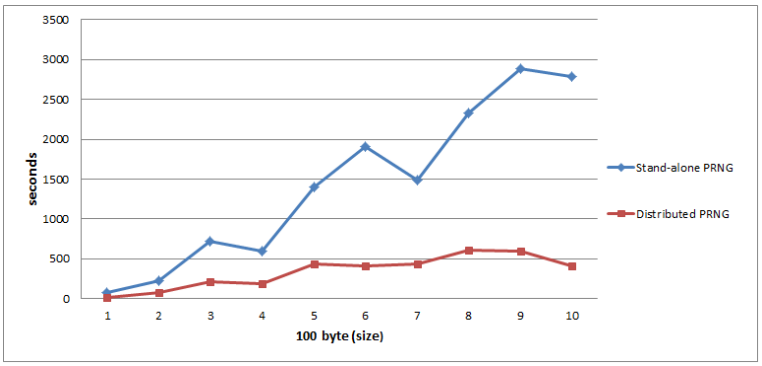

Fig. 3. A Comparison of Stand-alone Linux PRNG and our Distributed PRNG

From the Fig.3, we can see that our proposed is faster than the stand-alone Linux PRNG dev/random. We also can see that the time for generating pseudo-random number from dev/random does not always increases progressively with the output size. That is due to the unpredictability of the event entropy which imply a stronger security and robustness than dev/urandom which repeatedly use the pool entropy without enough update input for random events.

6 Conclusion and future works

In this paper, we proposed a distributed pseudo-random number generator based on Linux kernel and its PRNG. After that, we provide a solution for using the proposed PRNG to do the distributed random data perturbation which can be used to preserve the data privacy before using the cloud database service. The future direction should be modifying the our PRNG to make it more efficient and secure, we may try to use of a newer hash function, for example SHA-3 or AES to do the extracting output. It would require a significant change of the design, and an investigation of Linux PRNG.

References

1. L. Blum, M. Blum, and M. Shub. *Comparison of two pseudo-random number generators.* pp. 61-78, New York, Plenum Press, 1983.
2. Boaz Barak, Ronen Shaltiel, Eran Tromer: True Random Number Generators Secure in a Changing Environment. CHES 2003: 166-180.

3. Boaz Barak, Shai Halevi: A model and architecture for pseudo-random generation with applications to /dev/random. ACM Conference on Computer and Communications Security 2005: 203-212.
4. Mihir Bellare, John Rompel: Randomness-Efficient Oblivious Sampling FOCS 1994: 276-287
5. B. Chor, S. Goldwasser, S. Micali and B. Awerbuch. *Verifiable Secret Sharing and Achieving Simultaneity in The Presence of Faults.* In Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science, pp. 383-395, 1985.
6. D. Davis, R. Ihaka, and P. Fenstermacher. Cryptographic Randomness from Air Turbulence in Disk Drives. In Proceedings of Advances in Cryptology ECRYPTOE4, volume 839 of LNCS, pages 114-20,1994.
7. C. Dwork. *Differential privacy: a survey of results.* In Proc. of the 5th Intl Conf. on Theory and Applications of Models of Computation, TAMC, pp. 1-19, 2008.
8. Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergnaud and Daniel Wichs. *Security analysis of pseudo-random number generators with input: /dev/random is not robust*, Proceedings of the 2013 conference on Computer & communications security, pp. 647-658, USA, 2013
9. C. Ellison. *IEEE. P1363 Appendix E Cryptographic Random Numbers*, 1995. Available at: http://theworld.com/ cme/P1363/ ranno.html [online; accessed 2009].
10. Zvi Gutterman, Benny Pinkas, Tzachy Reinman: Analysis of the Linux Random Number Generator. Proc. of IEEE Security and Privacy, pp. 371-385, 2006
11. P. Gutmann. *Cryptographic Security Architecture Design and Verification,* 2004. ISBN 978-0-387-95387-8.
12. H. Krawczyk. *How to predict congruential generators.* In Advances in Cryptology-CRYPTOE9, volume 435 of LNCS, pages 138-53. Springer, 1990.
13. J. Krhovjak, J. Kur, V. Lorenc, V. Matyas, P. Pecho, Z. Riha, J. Staudek, P. Svenda, and Zizkovsky. *Smartcards, final report for the Czech National Security Authority*, December 2008.
14. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of The Art of Computer Programming. Addison-Wesley, third edition, 2001.
15. D. H. Lehmer. *Mathematical methods in large-scale computing units.* In Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery, Cambridge, MA, 1949, pages 141-146, Cambridge, MA, 1951. Harvard University Press.
16. *The LavaRnd Random Number Generator,* 2000. Available at: http://www.lavarnd.org/ [online; accessed 2009].
17. P. Lacharme. Post-Processing Functions for a Biased Physical Random Number Generator. Proc. of FSE'2008, pp.334 342, 2008.
18. T. G. Lewis and W. H. Payne. *Generalized feedback shift register pseudorandom number algorithm.* Journal of the ACM, 20(3):456-468, July 1973.
19. P. Lacharme, A. Rock, V. Strubel, and M. Videau, *The linux pseudorandom number generator revisited.* Cryptology ePrint Archive, Report 2012/251, 2012.
20. M. Matsumoto and Y. Kurita. *Twisted GFSR generators.* ACM Transactions on Modeling and Computer Simulation, 2(3):179-194, 1992.
21. M. Matsumoto and T. Nishimura. Mersenne Twister. *A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator.* ACM Transactions on Modeling and Computer Simulation (TOMACS), 8(1):3-30, 1998.
22. Matt Mackall and Theodore Tsfo. random.c A strong random number generator, 2009. /driver/char/random.c in Linux Kernel 2.6.30.7, http://www. kernel.org/.
23. A. Shamir. *On the generation of cryptographically strong pseudo-random sequences.* In Proc. ICALP, pages 544-550. Springer, 1981.
24. T. Ts'o. random.c Linux kernel random number generator. http://www. kernel.org