Generation of reliable randomness via social phenomena

Roberto De Virgilio and Antonio Maccioni

Dipartimento di Ingegneria
Università Roma Tre, Rome, Italy
{dvr,maccioni}@dia.uniroma3.it

Abstract

Randomness is a hot topic in computer science due to its important implications such as cryptography, gambling, hashing algorithms and so on. Due to the implicit determinism of computer systems, randomness can only be simulated. In order to generate reliable random sequences, IT systems have to rely on hardware random number generators. Unfortunately, these devices are not always affordable and suitable in all the circumstances (*e.g.,* personal use, data-intensive systems, mobile devices, etc.). Human-computer interaction (HCI) has recently become bidirectional: computers help human beings in carrying out their issues and human beings support computers in hard tasks. Following this trend, we introduce RandomDB, a database system that is able to generate reliable randomness from social phenomena. RandomDB extracts data from social networks to answer random queries in a flexible way. We prototyped RandomDB and we conducted some experiments in order to show the effectiveness and the advantages of the system.

1 Introduction

Randomness has been studied in many fields, from philosophy to psychology, from physics to social sciences. Analogously, mathematics and computer science have studied randomness under many aspects. In this context, randomness is considered to be the extent that allows us to obtain numbers and sequences of numbers (generally referred to as *random data*) in a non-predetermined and unpredictable manner. It means that every single number is equally probable to be drawn and the completion of a sequence cannot be inferred by correlation with previous sequences. The generation of random data has to be really accurate and therefore it is performed by complex agents, the so-called *Random Number Generator*s (*RNG*s). They allow computer systems to reach the "realism", precluded by deterministic procedures. Although at first glance the generation of random numbers can appear as simple as throwing a dice, it is a sensitive task for automatic agents. In fact, automatic agents are deterministic systems able to generate data only by processing machine-instructions and computing formulas. In case, it is possible to provide pseudo-randomness by executing an algorithm with a *secret* input, called *seed*. It means that, despite the user perception of a random generation, the randomness is only simulated. Moreover, the choice of a seed is a hard task because, if it is revealed, random data becomes predictable.

Researchers and companies are still investigating ways to improve both *RNG*s and the choice of effective seeds. Most of the generators are *Pseudo Random Number Generators* (*PRNG*s) able to reach only pseudo-randomness. In critical applications a *PRNG* is not good enough (and therefore not even allowed in some cases such as gambling applications). They can be cryptographically insecure, present a poor dimensional distribution, suffer some form of periodicity and use an unbounded size of memory. Therefore, using a *True Random Number Generator* (*TRNG*) is often required. Even if there is no exact way to determine if a *RNG* is *true* or *pseudo*, the *TRNG*s are those that gather information from entropic sources without using any sort of algorithm. For instance, they extract information from physical phenomena such as radioactive decay, thermal noise, micro-states of atoms and molecules. *TRNG*s are often implemented in expensive hardware devices and have a lower throughput than *PRNG*s. Therefore different kind of true random generators would be desiderale.

Motivation. HCI features are becoming more and more prominent in many computer science's areas. Unaware human decisions have recently became part of algorithms and computational processes in order to help computers in solving hard computational operations. The *reCAPTCHA* [15] project digitalizes books, newspapers and old time radio shows by exploiting the *CAPTCHA*[1] tasks, which were initially proposed for distinguishing human beings from computer bots. *Duolingo*[2] aims at translating the Web pages using language course exercises. Micro-tasks of any kind are widely managed and solved through the use of crowd-sourcing and collaborative platforms such as *Mechanical Turk Machine*[3]. In the database area, *CrowdDB* [5] uses people for answering queries. Following this trend, we try to solve the problem of reliable random number generation in a crowd-like way. Whereas a single human being shows a high degree of determinism when generating a random sequence of numbers [13], studies made by *sociophysics* (see [6] for survey) show that the society is a chaotic system and thus, some social phenomena follow entropic and physical behavior that are not ruled by determinism. Even entropy definitions such as the one in the second law of thermodynamics are likened to psychology [8] and to economics [3].

Contribution. From the motivations above, we can consider the collective behavior an entropic source and consequently, social networks (*e.g.,* Facebook, Twitter, Flickr, etc.) the virtual places where social phenomena take place. Nowadays social networks constantly produce huge amounts of data that reflect and capture the behavior of millions of people. Therefore, on the one hand, these social data comprise entropic data; on the other hand, due to the significant amount of data, they need a persistent storage, *i.e.* DBMS technology. For this purpose we propose a database system, called RandomDB, that, by exploiting social data, produces "good" randomness overcoming many drawbacks of existing *RNG*s. Therefore, RandomDB can be used as a reliable *RNG*, replacing the current state of the art.

[1] Completely Automated Public Turing test to tell Computers and Humans Apart
[2] http://duolingo.com/
[3] http://www.mturk.com/

Organization. The paper is organized as follows: in Section 2 we discuss related work, in Section 3 we overview the RandomDB showing the architecture and the system at work. In Section 4 we show the assessments of a developed prototype. Finally, we sketch conclusions and future evolutions in Section 5.

2 Related Work

*RNG*s are commonly classified into *PRNG*s and *TRNG*s. Famous *PRNG*s are the Linear Congruential Generator, Lehmer Random Number Generator, the Mersenne Twister [10], Blum Blum Shub, CryptGenRandom[4] (a.k.a. Microsoft CAPI) and Yarrow. *PRNG*s are easily predictable [2] and therefore in many critical applications we are forced to use systems from the plethora of *TRNG*s. Random.org[5] exploits atmospheric noise and offers random data through Web APIs. Lavarnd[6] selects a true random seed by taking pictures of floating material in lava lamps and then uses a *PRNG* to generate random data. HotBits[7] employs radioactive decays. Intel DRNG[8] generates random data at a high-rate from the thermal noise within the silicon. Finally, Protego[9] exploits quantum physics and GRANG [14] computes the thermal noise together with a Poisson arrival time for better efficiency. The main disadvantages of these methods are the high cost of ad-hoc devices and, often, their inability to reach an expected random data output rate. To this aim, work has been done in order to extract randomness from other contexts (*e.g.,* [12,4,7]). In [12] and [4] the authors exploit the entropy generated by human beings using mouse and keyboard. Since these methods are not always applicable and present dangerous vulnerabilities [16], the work in [7] aims to produce pseudo-random data with the conjunct use of extractors and people playing videogames.

3 System Overview

In this section we sketch how RandomDB works and overview its architecture.

System Architecture. As shown in Fig. 1, RandomDB is composed by three main components: (i) the *Query Processor* (QP), (ii) the *Mapper* (MP) and (iii) the *Generator* (GN). The MP and GN components are supported by a DBMS to store *Data* and *Metadata*.

In particular the *Data store* contains the social data concerning social phenomena. In a start-up phase, the administrator selects all data sources from which to extract all social data. Then such data are stored in a database with respect to a certain source domain and the *Metadata store* will contain all information useful to map source domains and target domains, as we will describe

[4] http://technet.microsoft.com/en-us/library/cc962093.aspx
[5] http://www.random.org/
[6] http://www.lavarnd.org/
[7] http://www.fourmilab.ch/hotbits/
[8] http://software.intel.com/
[9] http://www.protego.se/

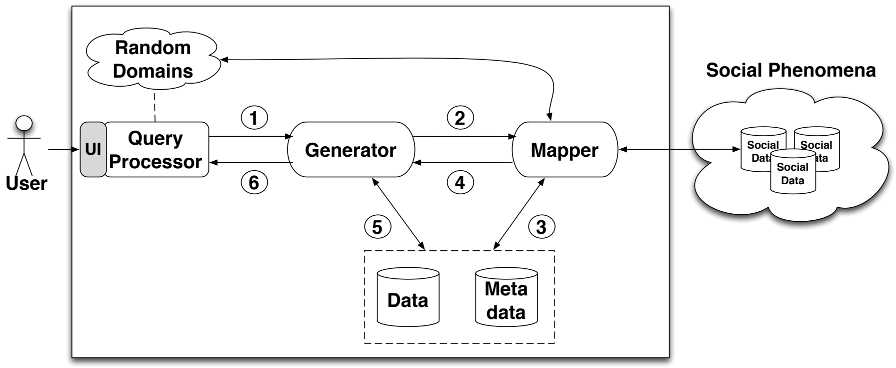


Fig. 1. RANDOMDB architecture.

later. The QP component provides an interface to interact with the *external world*.

System At Work. The working process is to map data from a *source* domain to a *target* domain. Source domains refer to data from social phenomena[10]. For example, let us consider people expressing preferences on a picture or on a topic (*i.e.* the `iLike` feature of many social networks); in this case the source domain S is represented by data that measures this liking (*e.g.,* natural numbers). The target domains correspond to random data domains that users and applications require. A target domain is associated to a query (*random query*) to perform in RANDOMDB. For example a *rand*()-like function of the C programming language is a random query that can be submitted to RANDOMDB. In this case the target domain refers to all real numbers in the range $(0, 1]$. Therefore, in order to generate a random real number in the range $(0, 1]$, our system has to use a mapping function $\mathcal{M}$ that maps a source S in $T = (0, 1]$, *i.e.* $\mathcal{M} : S \to T$. A user or an agent, through a *random query*, selects to generate random data in a particular target domain (*e.g.,* a real number, in case in a particular range). Then, the request is transferred to the GN component (*i.e.* Fig. 1, step ①). GN is responsible to generate the final random data fitting the user request. Such component has a central role in RANDOMDB architecture since it communicates with all the other components. GN communicates with MP (*i.e.* Fig. 1, step ②) to know which mapping has to be computed from one or more source domains associated to the selected target domain. MP is the component which is aware of the source domains and it is in charge of mapping them to the target domains. Such mappings are implemented in stored procedures on the underlying data store as will be explained in the next Section 4. MP exploits metadata (*i.e.* Fig. 1, step ③) in order to return a query to GN; such query will be executed over the underlying social data stored in our database (*i.e.* Fig. 1, step ④ and step ⑤). The procedure selects social data of interest and then, in case, applies some

[10] In this paper we do not delve into considerations about what is a social phenomenon and when it can be considered *entropic*.

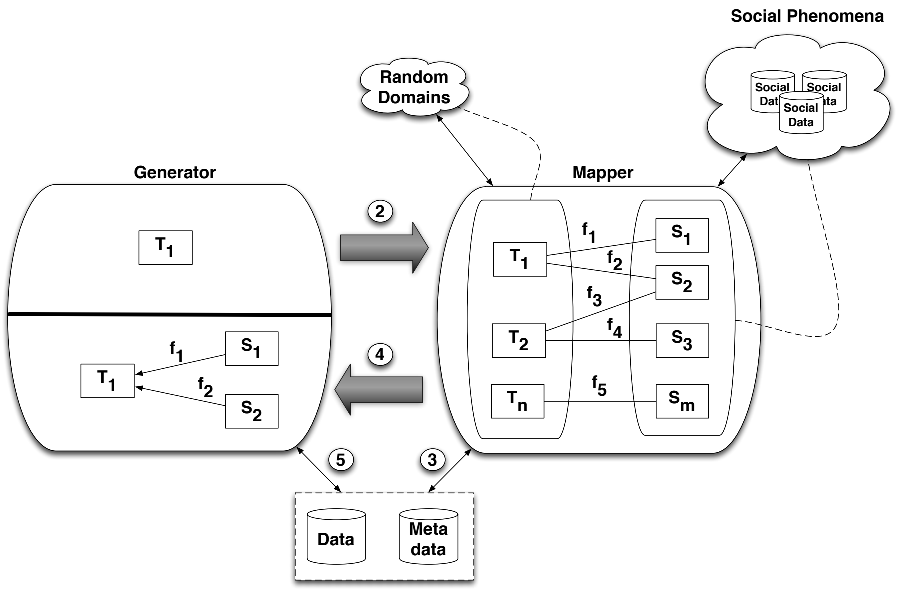


Fig. 2. RandomDB generation process.

transformations on such data. A transformation could be a simple operation, *e.g.,* a casting operation, or a more complex task, *e.g., mod* operation, to provide data correctly in the range of the target domain. In this last example, we are aware that a transformation can ruin the uniform distribution of the target domain interval. In the case of the mod, we have to be sure that the source domain interval is much larger than the target domain interval, *e.g.,* we cannot map a natural number in the range $[0, 1]$ into a natural number in the range $[0, 2]$. This *intelligence* is already embedded in MP that checks statistical metadata of the social data such as variance, covariance, standard deviation, minimum and maximum values, etc. Finally, the random data is returned to QP that will output the result (*i.e.* Fig. 1, step ⑥).

Fig. 2 sketches the working process of both MP and GN. Let us consider several target domains, *i.e.* $T_1, T_2, \ldots, T_n$ and source domains, *i.e.* $S_1, S_2, \ldots, S_m$. A target domain can be associated to one or more source domains, *e.g.,* a real number within $(0, 1]$ can be extracted from different phenomena. In this case we have a target domain, *i.e* T_1, corresponding to the set of real numbers within the range $(0, 1]$ and two source domains, *i.e* S_1 and S_2, corresponding for instance, to a Facebook phenomena and a Flickr phenomena, respectively. Given the mapping $\mathcal{M}' : \langle S_1, S_2 \rangle \rightarrow T_1$, to generate an element of T_1 the mapping can use an element of S_1 or of S_2, or a composition of elements from both S_1 and S_2.

Use of the System. RandomDB can be configured on top of common DBMSs and used in two scenarios. In the former is used by a human user that wants to get a random data upon request, while in the latter case is used embedded into an application where QP is a programmable interface to allow an automatic communication between RandomDB and the application itself (*i.e.* API). In this last scenario, contrary to common *RNG*s, we can exploit RandomDB to work with the active domain of existing database instances (the actual content of the database). The active domain of the database can, even dynamically, form target domains. For example, if we have a table containing cities, RandomDB would be able to directly retrieve randomly a city. This operation can be simply computed by enumerating tuples (or values) and after that, generating a number in the range $[1, \ldots, t]$, where t is the number of objects in the range of the target domain.

4 System Evaluation

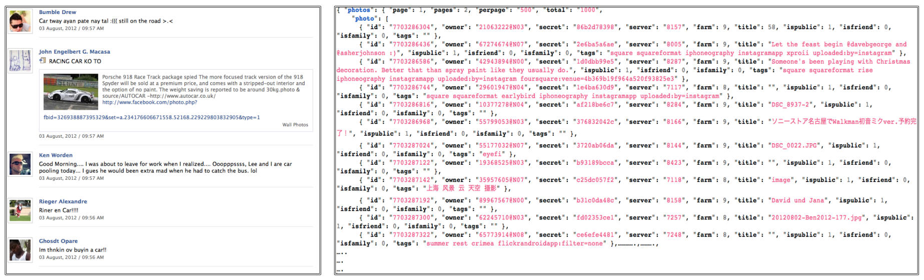
This section evaluates the prototype of RandomDB that we have implemented. First, we explain how it was created and then we provide the evaluation in terms of efficiency and effectiveness. This also shows the feasibility to implement randomness generation processes using a DBMS.

Implementation. RandomDB is implemented in Java and exploits a procedural language for SQL procedures to implement all mappings and transformations. In particular PL/pgSQL, since we used PostgreSQL 9.1 as RDBMS. Java is used to implement a set of wrappers and different programmatic accesses to social phenomena in several social networks. At the moment, we consider the following social networks.

- *Flickr*[11]: we extracted the most recent photos (500 at a time) via the Flickr API. Let us consider Fig. 3 that, on the left side, shows an extract from Flickr and, on the right side, shows the retrieved information. They contain lot of different information that can be considered random. In particular, we extract values from the fields `secret`, `owner` and `title`. In future work we can extend the extraction by considering also information extracted from the pixel values of the photos.
- *Twitter*[12]: we used the tags associated to the photos extracted by Flickr to search the most recent 100 tweets.
- *Facebook*[13]: similarly to Twitter, we used the tags associated to the photos extracted by Flickr to search the most recent 200 posts.

Fig. 4 shows the schema we implemented to test our system. In particular we have three main relations: (i) Domains, (ii) Mappings and (iii) SocialData. The first two tables belong to the *Metadata store* while the third to the *Data store*. The relation Domains collects all meta information about the domains (both source and target) considered in RandomDB. In particular we store an identifier, *id* that is the key of the relation, the *name* of the domain and the *domain_type*, whose value is `s` (for source) or `t` (for target). The attribute *data_type*

[11] http://www.flickr.com/services/api/
[12] https://dev.twitter.com/
[13] http://developers.facebook.com/



Bumble Drew
Car tway ayan pate nay tal :(((still on the road >.<
03 August, 2012 / 09:57 AM

John Engelbert G. Macasa
RACING CAR KO TO

Porsche 918 Race Track package spied The more focused track version of the 918 Spyder will be sold at a premium price, and comes with a stripped-out interior and the option of no paint. The weight saving is reported to be around 30kg.photo & source/AUTOCAR –http://www.autocar.co.uk/
http://www.facebook.com/photo.php?

fbid=326938887395329&set=a.234176606671558.52168.229229803832905&type=1
Wall Photos

03 August, 2012 / 09:57 AM

Ken Worden
Good Morning.... I was about to leave for work when I realized.... Ooopppssss, Lee and I are car pooling today... I gues he would been extra mad when he had to catch the bus. lol
03 August, 2012 / 09:57 AM

Rieger Alexandre
Riner en Car!!!!
03 August, 2012 / 09:56 AM

Ghosdt Opare
Im thnkin ov buyin a car!!
03 August, 2012 / 09:56 AM

```
{ "photos": { "page": 1, "pages": 2, "perpage": "500", "total": "1000",
    "photo": [
      { "id": "7703286304", "owner": "21063222@N03", "secret": "86b2d78398", "server": "8157", "farm": 9, "title": 58, "ispublic": 1, "isfriend": 0,
"isfamily": 0, "tags": "" },
      { "id": "7703286436", "owner": "67274674@N07", "secret": "2e6ba5a6ae", "server": "8005", "farm": 9, "title": "Let the feast begin @davebgeorge and
@asherjohnson :)", "ispublic": 1, "isfriend": 0, "isfamily": 0, "tags": "square squareformat iphoneography instagramapp xproii uploaded:by=instagram" },
      { "id": "7703286586", "owner": "42943894@N00", "secret": "1d0dbb99e5", "server": "8287", "farm": 9, "title": "Someone's been playing with Christmas
decoration. Better that than spray paint like they usually do.", "ispublic": 1, "isfriend": 0, "isfamily": 0, "tags": "square squareformat rise
iphoneography instagramapp uploaded:by=instagram foursquare:venue=4b369b19f964a520f93825e3" },
      { "id": "7703286744", "owner": "29601947@N04", "secret": "1e4ba630d9", "server": "7117", "farm": 8, "title": "", "ispublic": 1, "isfriend": 0,
"isfamily": 0, "tags": "square squareformat earlybird iphoneography instagramapp uploaded:by=instagram" },
      { "id": "7703286816", "owner": "10377278@N04", "secret": "af218be6c7", "server": "8284", "farm": 9, "title": "DSC_8937-2", "ispublic": 1,
"isfriend": 0, "isfamily": 0, "tags": "" },
      { "id": "7703286968", "owner": "55799053@N03", "secret": "376832042c", "server": "8166", "farm": 9, "title": "ソニーストア名古屋でWalkman初音ミクver.予約完
了!", "ispublic": 1, "isfriend": 0, "isfamily": 0, "tags": "" },
      { "id": "7703287024", "owner": "55177032@N07", "secret": "3720ab06da", "server": "8144", "farm": 9, "title": "DSC_0022.JPG", "ispublic": 1,
"isfriend": 0, "isfamily": 0, "tags": "eyefi" },
      { "id": "7703287122", "owner": "19368525@N03", "secret": "b93189bcca", "server": "8423", "farm": 9, "title": "", "ispublic": 1, "isfriend": 0,
"isfamily": 0, "tags": "" },
      { "id": "7703287142", "owner": "35957605@N07", "secret": "c25dc057f2", "server": "7118", "farm": 8, "title": "image", "ispublic": 1, "isfriend": 0,
"isfamily": 0, "tags": "上海 风景 云 天空 摄影" },
      { "id": "7703287192", "owner": "89967567@N00", "secret": "b31c0da48c", "server": "8158", "farm": 9, "title": "David und Jana", "ispublic": 1,
"isfriend": 0, "isfamily": 0, "tags": "" },
      { "id": "7703287300", "owner": "62245710@N03", "secret": "fd02353ce1", "server": "7257", "farm": 8, "title": "20120802-Ben2012-177.jpg", "ispublic":
1, "isfriend": 0, "isfamily": 0, "tags": "" },
      { "id": "7703287322", "owner": "65773914@N08", "secret": "ce6efe4481", "server": "7248", "farm": 8, "title": "", "ispublic": 1, "isfriend": 0,
"isfamily": 0, "tags": "summer rest crimea flickrandroidapp:filter=none" },..........,.........,
....
...
...
```

Fig. 3. Extraction of social data from Flickr.

indicates which type of data is associated to the domain. Then we store an attribute *range*, indicating the range of the domain.

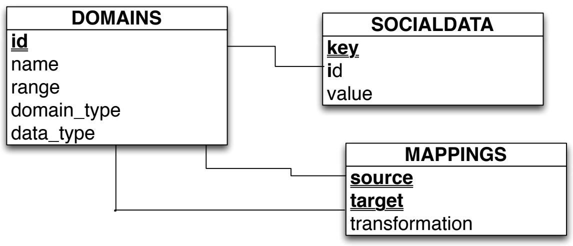


Fig. 4. Database schema in RandomDB.

The relation Mappings collects all mappings between domains; in particular the attributes *source* and *target* refer to identifiers in Domains whose types have values `s` and `t`, respectively. The attribute *transformation* stores an identifier of a stored procedure that is needed to retrieve correctly the requested random data. Finally, we have the relation SocialData containing all data values extracted from the social phenomena. In particular each entry of SocialData provides an auto-increment primary key, *i.e.* *key*, an identifier *id* corresponding to the identifier in the relation Domains, *i.e.* it is a foreign key, and the data value in the domain, *value*. For instance, Fig. 5 shows an example of instance corresponding to the schema of Fig. 4. In this case we have four source domains extracted from Flickr, Twitter and Facebook. Then we define several mappings to generate random data. Let us consider the mapping $\mathcal{M}' : \langle S_1, S_2 \rangle \rightarrow T_1$. In this case we can generate random data for the target domain `Target1` from Facebook and/or Flickr. Each mapping is processed by a PL/pgSQL procedure to generate the corresponding queries. In this case we have to select the first available value from the source domains (*i.e.* $\mathtt{S}_1$ and $\mathtt{S}_2$), as we can see in the following query:

Domains

id	name	range	domain_type	data_type
S_1	Flickr	[-32768, 32767]	s	smallint
S_2	Facebook	[-999999, 999999]	s	int
S_3	Flickr	[a,z]	s	char
S_4	Twitter	[0, 4294967295]	s	uint
T_1	Target1	(0,50]	t	int
T_2	Target2	[a,z]	t	char
T_3	Target3	(0, 1]	t	real
...	...	...	...	...

SocialData

key	id	value
7898	S_4	78999
7899	S_2	770328
8000	S_1	124
8001	S_3	b
8002	S_1	23456
8003	S_2	-1793,89
...	...	...

Mappings

source	target	transformation
S_1	T_1	f_1
S_2	T_1	f_2
S_2	T_2	f_3
...	...	...

Fig. 5. An instance of tables in RandomDB.

```
SELECT id, value
FROM SocialData
WHERE id = S1 OR id = S2
ORDER BY key
LIMIT 1
```

Considering the instance tables in Fig. 5, the query returns the pair `<S`$_2$`, 770328>`. Then, another query selects the *transformation* that we have to execute on the retrieved *value* for the given mapping:

```
SELECT transformation
FROM Mappings
WHERE source = S2 and target = T1
```

The identifier f_2 is returned to individuate a built-in operation or another stored procedure. For instance, let us suppose that f_2 `= CAST(MOD(value,50) as int)`. Now the final query can compute the transformation f_2 on the value 770328:

```
SELECT CAST(MOD(770328,50) AS INT)
```

The answer 28 is returned to the user. Once we generated the random data, we delete the entry in the table SocialData used for the generation. Our system will enrich periodically the *Data store* by extracting new values from the social phenomena that will be appended in the table SocialData. Note that the extracted data in the *Data store* are sequentially ordered on the exact time of generation. In this way the simulation of the system is the same as if the whole process was computed online.

TRNG	**Throughput**
Protego	16 KB/s (*)
GRANG	50 MB/s (*)
Intel DRNG	500 MB/s (*)
Hotbits	100 B/s (*)
RandomDB	400 KB/s

$*$ Declared by the vendor

Table 1. Throughput of *TRNG*s.

Efficiency Evaluation. The efficiency of RandomDB is based on two factors: (i) extraction of social data and (ii) query processing on the database. The former is dependent from the bandwidth and access policy of the social networks. The latter is based on the physical design of the database and on the speed of the machine where RandomDB is deployed. We compared the throughput of RandomDB against the most used *TRNG*s in Table 1. We used the declared rates by the vendors because as said before, these systems are commercial (and expensive), so we couldn't test them by ourself. We did not consider the *PRNG*s for the efficiency, which however is similar to RandomDB (*i.e.* mostly it depends on the the power of the machines). We used a commodity laptop with a dual core 2.5 GHz Intel and 4 GB of memory, running Linux Ubuntu. We measured the throughput of RandomDB posing a loop of random queries for the integer numbers target domain.

Under this aspect, we perform better than some of the *TRNG*s. Note that we can improve the performance if we use a faster machine. However, the high rate throughput *TRNG*s are expensive hardware devices.

Effectiveness Evaluation. Evaluating the effectiveness of random data generation is a very complex task, probably even harder than the generation itself. Currently, empirical statistical tests are considered the best way to evaluate a *RNG* [11]. Since *"Building a RNG that passes all statistical tests is an impossible dream"*, as famously stated by Pierre L'Ecuyer, the goodness of a *RNG* is evaluated on the number of passed tests among a battery of many statistical significance tests [9]. These tests define, on the next random data to be generated, hypotheses enclosed in confidence intervals $[\alpha, 1-\alpha]$, where α is usually set to 0.05 or to 0.01. The hypotheses are based on criteria over previous generated data. For example, a criteria can consider if the sum of a sequence of numbers is constant. Therefore, for a better evaluation one should compute a set of different tests over random generated sequences. The result of a test is summarized by the *p-value* which represents a probability (in fact *p-value* $\in [0,1]$) to measure the support for randomness hypothesis. The test is considered passed, *i.e.* the sequence is not predictable on the basis of the given hypotheses, if $|p\text{-}value - 1| \geq \alpha$, *i.e.* $abs(p\text{-}value - 1) \geq \alpha$, otherwise the test is failed. Note that two different sequences generated with the same *RNG* can return different *p-values*. This is the reason why, for the same *RNG*, the tests are executed many times over different sequences. Among the existing statistical tests for *RNG*s, the Diehard battery tests of George Marsaglia[14] and the NIST [1] test collection

[14] http://www.stat.fsu.edu/pub/diehard/

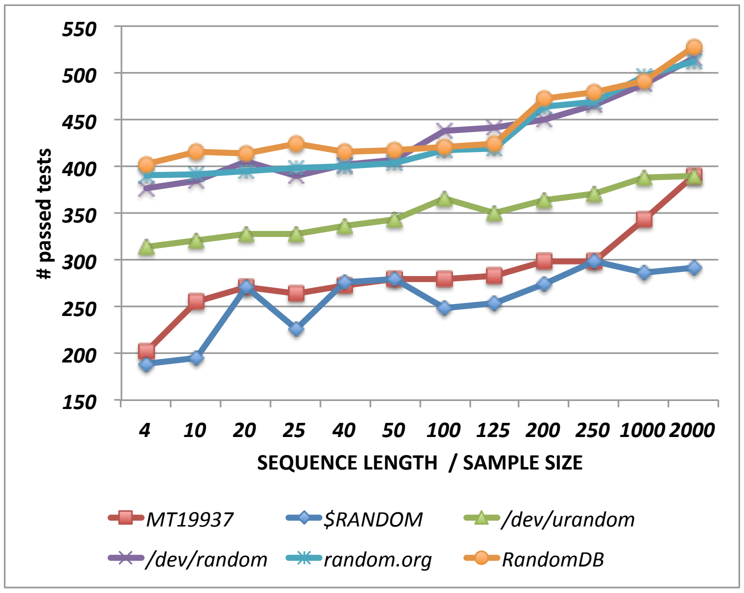


Fig. 6. RNGs comparison with Dieharder's tests.

are the most common. They both implement the principles explained above and, in particular, the last one is becoming the de-facto standard battery test since it is required in many critical applications. In order to assess RandomDB we used the Dieharder test suite[15] that is included in the GNU Scientific Library (GSL). It comprises both the Marsaglia's tests (*e.g.,* the *birthday spacings test* that, based on the birthday paradox, checks if the spacings between the numbers are asymptotically exponentially distributed) and some of the NIST tests (*e.g.,* the *serial tests* that check if the overlappings of patterns across a sequence are equally probable). The suite can exploit *RNG*s of the GSL, or takes in input a sequence of bits or numbers. We conducted two different experimental campaigns, the first to compare RandomDB against other *RNG*s (both *pseudo* and *true*) and the second to evaluate the probability distribution of the elements in a target domain. For the first campaign we compared RandomDB with the MT19937 variant of the Mersenne Twister algorithm [10], the Linux shell random number generator ($RANDOM) and the Linux random number generator /DEV/URANDOM as *PRNG*s. We also used the RANDOM.ORG[16] and the Linux /DEV/RANDOM as *TRNG*s. With every *RNG*, we generated 10 sequences of 1000 numbers, 10 of 5000 numbers and 10 of 10000 numbers. For a more comprehensive evaluation, each sequence has been tested with 4 different sample sizes (5, 40, 100, 250). The ratio between the sequence length and the sample size (*i.e.* $\frac{seq.\ length}{sample\ size}$) defines a test case. This ratio is also an indicator of the complexity

[15] http://www.phy.duke.edu/~rgb/General/dieharder.php
[16] http://www.random.org/

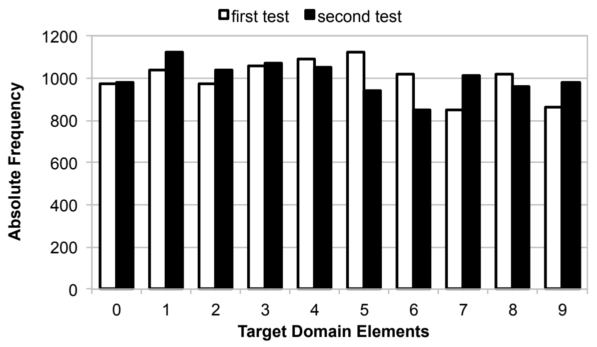


Fig. 7. Frequency distribution for [0,9] integers in RandomDB.

to predict a random number since a larger sample size brings a smaller *p-value*, for more details see [1]. Every test case has been tested with each of the 57 different tests within the Dieharder library and with $\alpha = 0.005$ (so making the tests harder than the default ones with $\alpha = 0.05$). Then, we counted the times that an *RNG* test case passed the verification out of the 570 launched tests, as shown in Fig. 6.

We can see that in general the *TRNG*s perform better than the *PRNG*s. RandomDB performs almost always better than the others. For sure, we can say that it exhibits a reliable behavior, as if it is a *TRNG*. The second campaign has been conducted to see if, in RandomDB, every number is equally probable. We computed the frequency distribution of the target domain $[0, 9]$. We made two extractions (*i.e.* we call them *first test* and *second test*) of 10000 numbers and we counted the frequency. In Fig. 7 we show the frequency of each number. It shows that the probability of all the numbers is uniform, i.e. we can consider every element of the range equally probable.

5 Conclusions and Future Work

In this paper we presented RandomDB, a database system to generate reliable randomness via social phenomena. It exploits human-computer interaction (HCI) and the data generated in social networks. RandomDB is highly flexible and easily embeddable in software applications. We prototyped RandomDB and we conducted some experiments in order to show the effectiveness and the advantages of the system. Experiments showed that RandomDB is reliable as a *TRNG* in generating random data. Our first design of RandomDB is basically constructed upon social Web data. We are individuating other fields that will be used to "feed" RandomDB. For example data from sensor networks, since they are more and more deployed on cities and environments to measure natural phenomena. This opens future directions to data-recycling in many other contexts. Moreover we are considering to introduce crowd sourcing techniques to improve the effectiveness of our system. From a practical point of view, we are working on the prototype in order to test it with the official NIST benchmark. In this way RandomDB can be proposed for a wide-usage.

References

1. Bassham, III, L.E., Rukhin, A.L., Soto, J., Nechvatal, J.R., Smid, M.E., Barker, E.B., Leigh, S.D., Levenson, M., Vangel, M., Banks, D.L., Heckert, N.A., Dray, J.F., Vo, S.: A statistical test suite for random and pseudorandom number generators for cryptographic applications. SP 800-22 Rev. 1a. (2010)
2. Boyar, J.: Inferring sequences produced by pseudo-random number generators. J. ACM 36(1), 129–141 (1989)
3. Chen, J.: The Physical Foundation of Economics: An Analytical Thermodynamic Theory. World Scientific Publishing Company (2005)
4. Dorrendorf, L., Gutterman, Z., Pinkas, B.: Cryptanalysis of the random number generator of the windows operating system. ACM Trans. Inf. Syst. Secur. 13(1) (2009)
5. Franklin, M.J., Kossmann, D., Kraska, T., Ramesh, S., Xin, R.: CrowdDB: answering queries with crowdsourcing. In: SIGMOD. pp. 61–72 (2011)
6. Galam, S.: Sociophysics: A Physicist's Modeling of Psycho-political Phenomena. Springer (2012)
7. Halprin, R., Naor, M.: Games for extracting randomness. In: SOUPS (2009)
8. La Cerra, P.: The First Law of Psychology is the Second Law of Thermodynamics: The Energetic Evolutionary Model of the Mind and the Generation of Human Psychological Phenomena. Human Nature Review 3, 440–447 (2003)
9. L'Ecuyer, P., Simard, R.J.: TestU01: A C library for empirical testing of random number generators. ACM Trans. Math. Softw. 33(4) (2007)
10. Matsumoto, M., Nishimura, T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. 8(1), 3–30 (1998)
11. Maurer, U.M.: A universal statistical test for random bit generators. J. Cryptology 5(2), 89–105 (1992)
12. de Raadt, T., Hallqvist, N., Grabowski, A., Keromytis, A.D., Provos, N.: Cryptography in openbsd: An overview. In: USENIX Annual Technical Conference, FREENIX Track. pp. 93–101 (1999)
13. Rapoport, A., Budescu, D.V.: Randomization in Individual Choice Behavior. Psychological Review 104(3), 603–617 (Jul 1997)
14. Saito, T., Ishii, K., Tatsuno, I., Sukagawa, S., Yanagita, T.: Randomness and genuine random number generator with self-testing functions. http://csrc.nist.gov/rng/rng2.html (2010)
15. Von Ahn, L., Maurer, B., McMillen, C., Abraham, D., Blum, M.: reCAPTCHA: Human-based character recognition via web security measures. Science 321(5895), 1465–1468 (August 2008)
16. Yilek, S., Rescorla, E., Shacham, H., Enright, B., Savage, S.: When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In: Proceedings of IMC 2009. pp. 15–27. ACM Press (Nov 2009)