ANALYSIS OF ANDROID RANDOM NUMBER GENERATOR

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Serkan Sarıtaş

May, 2013

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. A. Aydın Selçuk(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. İbrahim Körpeoğlu

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Sinan Gezici

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

ANALYSIS OF ANDROID RANDOM NUMBER GENERATOR

Serkan Sarıtaş
M.S. in Computer Engineering
Supervisor: Asst. Prof. Dr. A. Aydın Selçuk
May, 2013

Randomness is a crucial resource for cryptography, and random number generators are critical building blocks of almost all cryptographic systems. Therefore, random number generation is one of the key parts of secure communication. Random number generation does not guarantee security. Problematic random number generation process may result in breaking the encrypted communication channel, because encryption keys are obtained by using random numbers. For computers and smart devices, generation of random numbers is done by operating systems. Applications which need random numbers for their operation request them from the operating system they are working on.

Due to the importance of random number generation, this process should be analyzed deeply and cryptographically for different operating systems. From this perspective, we studied Android random number generation process by looking at the source codes and found that security of random number generation done by Android relies on the security of random number generation of Linux. Then we analyzed Android random number generator by modifying the kernel source code and applying some tests on its entropy estimator. Finally, we looked for possible weaknesses of random number generator during startup of Android devices.

Keywords: SecureRandom, random number generation/generators, Linux RNG, Android RNG, entropy estimator.

ÖZET

ANDROID RASSAL SAYI ÜRETECİNİN ANALİZİ

Serkan Sarıtaş
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Y. Doç. Dr. A. Aydın Selçuk
Mayıs, 2013

Rassallık, kriptoloji için çok önemli bir kavramdır ve rassal sayı üreteçleri, hemen hemen tüm kriptografik sistemlerde kullanılan temel yapı taşlarındandır. Bu nedenle, rassal sayı üretimi, güvenli iletişimin anahtar noktalarındandır. Rassal sayı üretimi güvenli iletişimin güvencesini vermez. Sorunlu rassal sayı üretim işlemi, zayıf şifreleme anahtarları oluşturacağından güvenli iletişim hatlarının kırılmasına sebep olabilir. Bilgisayarlar ve akıllı cihazlarda rassal sayı üretimi, işletim sistemleri tarafından gerçekleştirilir. Uygulamalar, çalışmaları esnasında ihtiyaç duydukları rassal sayıları, işletim sistemlerinden talep ederler.

Rassal sayı üretiminin çok hassas ve önemli bir süreç olmasından ötürü, bu sürecin farklı işletim sistemleri için derinlemesine ve kriptografik olarak incelenmesi gerekmektedir. Bu noktadan yola çıkarak, Android işletim sisteminin kaynak kodlarına bakarak rassal sayı üretim sürecini inceledik ve Android işletim sisteminin güvenli rassal sayı üretiminin Linux işletim sisteminin rassal sayı üretiminin güvenliğine bağlı olduğunu tespit ettik. Ardından Android işletim sisteminin çekirdeğinin kaynak kodlarını değiştirerek rassal sayı üretecini test ettik ve entropi tahminleri üzerinde farklı testler gerçekleştirdik. Son olarak, Android cihazların açılışı esnasında, rassal sayı üretimi merkezli ortaya çıkabilecek zayıflıkları araştırdık.

Anahtar sözcükler: SecureRandom, rassal sayı üretimi/üreteçleri, Linux Rassal Sayı Üreteci, Android Rassal Sayı Üreteci, entropi tahmini.

Acknowledgement

First of all, I am grateful to my thesis supervisor Asst. Prof. Dr. A. Aydın Selçuk for his supervision, support, encouragement, patience and suggestions for the completion of my thesis, from the beginning to the end.

I would also like to thank Assoc. Prof. Dr. İbrahim Körpeoğlu and Assoc. Prof. Dr. Sinan Gezici as my examining committee members.

Special appreciation and thanks to Elif Ahsen Tolgay and Ahmet Yükseltürk for reviewing the drafts.

I wish to extend my thanks to all friends and colleagues for their valuable help in the development of this thesis. I especially would like to thank my friends Şeyma Canik, Furkan Çimen, Bulut Esmer, Berna Girgin, Sayım Gökyar, Fatih Hafalır, Hasan Hamzaçebi, Mürsel Karadaş, Furkan Keskin, Ali Cahit Köşger, Nilgün Öz, Elif Ahsen Tolgay, Uğur Yılmaz, Ahmet Yükseltürk who have been on my side during my happy and difficult days in these last three years. I would like to express my gratitude to my dear friends in Bilkent Orienteering Club. Last but not the least, I should also thank to our dorm officer Nimet Abla. For today and for tomorrows, you will always be like a family to me.

I would also like to thank TÜBİTAK for providing financial support throughout my graduate study.

There are no words to express my gratitude to my family and my relatives. All can be said is that I am the most fortunate person to have such parents—Nejla and Kamil, a brother—Hakan, and his wife—Burcu, a sister—Hatice, aunts—Emine, Şefika and Aysel, uncles—Mehmet and Turan, and all relatives. I would like to express my special thanks to them for their sincere love, support and encouragement.

Contents

List of Figures

List of Tables

List of Codes

List of Algorithms

Chapter 1

Introduction

"Everything we do to achieve privacy and security in the computer age depends on random numbers."
—Simon Cooper

"Random numbers should not be generated with a method chosen at random."
—Donald Knuth

In this research, we investigate the random number generation process of Android OS running on the emulator.

Android OS is an open source project designed primarily for touchscreen devices. It is derived from the Linux OS which is also open source so their kernels are nearly the same. Although the libraries and basics of Android are written in C, application software running on an application framework which includes Java-compatible libraries is based on Apache Harmony. Android uses the Dalvik virtual machine with just-in-time compilation to run `Dalvik dex-code (Dalvik`

`Executable)`, which is usually translated from `Java bytecode`. Therefore, top-level implementation of random number process of Android is written in Java.

Android has `SecureRandom` class for generating cryptographically secure random numbers. This generation process is totally deterministic if input-seed is known. Input-seed comes from the kernel part of random number generation process. Therefore, in order to analyze the security of random number generator of Android, we examine Android kernel. Android uses slightly modified Linux kernel; but their random number generation processes are the same. Because of this reason, by analyzing Android RNG, we also analyze Linux RNG.

During the analysis process, first we download Android source code and detect the parts that are related to random number generation. After viewing these parts, we see that the main source comes from kernel. Then, we download Android kernel source code and search the same parts, specifically `random.c` class. In order to understand how the system works better, we modify this class so that some intermediate outputs give additional information about the random number generation process after examining the source code,.

Subsequently, we apply some tests on kernel to evaluate its reliability in calculation of entropy estimation, because the heart of the system lies on the entropy estimator; i.e. the calculated amount of the entropy decides the quality of the random number generator outputs. As it will be explained later, we could not find any obvious weakness of the estimator.

1.1 Background

Randomness is a crucial resource for cryptography, and random number generators are critical building blocks of almost all cryptographic systems. Therefore, random number generation is one of the key parts of secure communication. Problematic random number generation process may result in breaking the encrypted communication channel, because the encryption keys are obtained by using the

random numbers. For computers and smart devices, generation of random numbers is handled by operating systems because of the fact that obtaining truly random numbers from the physical sources is a costly method. Digital devices are fully deterministic machines but unpredictability is still required for cryptography, security, randomized algorithms, scheduling and networking; therefore, some modifications and additions are needed in order to construct random number generators using these machines.

There are basic requirements recommended by [6] that random number generators must hold even if the attacker knows the code of the generator, and/or has partial knowledge of the entropy used for refreshing the generator's state. These requirements can be listed as follows:

Pseudorandomness (Resilience) : The generator's output looks random to an observer with no knowledge of the internal state. This property must hold regardless of that observer's complete control over data, which is used to refresh the internal state.

Forward security : An adversary which learns the internal state of the generator cannot learn anything about previous outputs of the generator. In other words, past outputs of the generator must look like random to an observer, even if that observer learns the internal state afterwards.

Backward security (Break-in recovery) : An adversary which learns the state of the generator cannot learn anything about future outputs of the generator. Namely, future outputs of the generator looks random, even to an observer with knowledge of the current state. This property is satisfied by using sufficient entropy to refresh the generator's state.

Regarding forward security, note that generator must not leak any information about its previous states and outputs. In order to achieve this property, the methods which are easily calculated in forward direction but cannot be calculated in the reverse direction must be used. This property is named *one-wayness property* and hash functions are example of this family of functions. Backward security, on the other hand, cannot be satisfied for deterministic functions. Recall

that, software-based random number generation is just deterministic process and cannot provide backward security if used alone. In order to eliminate deterministic property of software generators, states of the generators must periodically be refreshed with sufficiently random external data.

As a concrete example to these properties, Linux RNG can be given. Linux RNG provides backward security by collecting entropy from several noise sources. These entropy sources are based on user activity events, such as pressing a key on a keyboard or moving a mouse, or system events which include interrupts and hard disk I/O. In order to satisfy forward security property, Linux RNG uses hash functions between state transitions.

1.2 Attacks on Random Number Generators

Problems in random number generators may cause very critical security flaws. In this section, critical attacks on random number generators will be listed.

1.2.1 Attack to Netscape Browser's SSL Implementation

SSL implementation of Netscape's Solaris 2.4 browser has weakness on random number generator as described in [7]. It was discovered that Netscape browsers generated SSL session keys using second, microsecond, process ID and parent process ID as seed, as shown in Code 1.1. After a seed is obtained, generating encryption key is a totally deterministic process as shown in Code 1.2. Therefore, security of the encryption scheme relies on the security of the seed. However, a seed is guessable for an attacker who has an account on the UNIX machine running the Netscape browser, and likewise for an attacker who does not have an account. Former group can learn process ID and parent process ID by simply logging into the system. Then, in order to learn time value, attacker just uses Ethernet sniffing tools to see precise time of each packet and by using this he can guess the time of day on the system running the Netscape browser to within

```
global variable seed;

RNG_CreateContext()
     /* Time elapsed since 1970 */
    (seconds, microseconds) = time of day;
    pid = process ID;  ppid = parent process ID;
    a = mklcpr(microseconds);
    b = mklcpr(pid + seconds + (ppid << 12));
     seed = MD5(a, b);

/* not cryptographically significant; shown for
   completeness */
mklcpr(x)
    return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);

/* a very good standard mixing function, source omitted */
MD5()
```

Code 1.1: The Netscape v1.1 seeding process: pseudocode. Only unknowns are *second*, *microsecond*, *pid* and *ppid* values for attacker.

a second. After that, he can easily guess microsecond value by brute-forcing—there are only one million possibilities. For the latter group, attack is more complicated. In particular, even though the *pid* and *ppid* are 15 bit quantities on most UNIX machines, the sum $pid + (ppid \ll 12)$ has only 27 bits, not 30. If the value of seconds is known, variable a has only 20 unknown bits, and variable b has only 27 unknown bits. This leaves, at most, 47 bits of randomness in the secret key, a far cry from the 128-bit security claimed by the domestic U.S. version.

An ironic aspect should be mentioned at this point. Unfortunate for Netscape, U.S. regulations prohibit the export of products incorporating strong cryptography. In order to distribute an international version of its browser overseas, Netscape had to weaken the encryption scheme to use keys of just 40 bits, which is even less than 47 bits of randomness in Netscape domestic version due to the weaknesses in the implementation.

```
RNG_GenerateRandomBytes()
    x = MD5(seed);
    seed = seed + 1;
    return x;

global variable challenge, secret_key;

create_key()
    RNG_CreateContext();
    tmp = RNG_GenerateRandomBytes();
    tmp = RNG_GenerateRandomBytes();
    challenge = RNG_GenerateRandomBytes();
    secret_key = RNG_GenerateRandomBytes();
```

Code 1.2: The Netscape v1.1 key-generation process: pseudocode. If seed is known, key generation is totally deterministic.

1.2.2 Attack to Kerberos v4 Session Keys

An attack similar to the one on Netscape was demonstrated in 1997, on the MIT implementation of the Kerberos 4.0 authentication protocol [1]. Kerberos Version 4 uses the UNIX random function to produce the random DES keys. Kerberos generates a random DES key by first seeding the random number generator with a seed chosen as in Algorithm 1.1, then it makes two calls to the random function to get 64 pseudorandom bits. 56-bit DES key is extracted from this 64-bit block. The random function relies on a 32-bit seed value to determine the internal state for generating the pseudorandom numbers. Thus, any sequence of numbers created by this random function, no matter how long they are, has an entropy of only 32 bits. Likewise the Kerberos session keys have an entropy of only 32 bits.

By improving the attack, attack complexity can be reduced to 2^{20}. As it was mentioned the only component of the seed that significantly changes between successive key generations is the microseconds value. This yields a key entropy of about 20 bits. Unlike the low-order 20 bits, the first 12 bits rarely change and are predictable; because the values other than microsecond values do not change very much. This can be seen graphically in Figure 1.1. As a result of this poor

```
time = time-of-day seconds since UTC 0:00 Jan. 1, 1970
pid = process ID of the Kerberos server process
keyCount = cumulative count of session keys generated
fTime = fractional part of time-of-day seconds since UTC
   0:00 Jan. I, 1970 in microseconds
hid = hostid of the machine on which the Kerberos server is
    running
seed = time ⊕ pid ⊕ keyCount ⊕ fTime ⊕ hid
Note that all values are 32-bits
```

Algorithm 1.1: Kerberos v4 seed generation algorithm. *fTime* is the most changing value, it determines unpredictability.

choice in seed values, given knowledge of the approximate time that a key was generated, there are only about 2^{20} (or approximately one million) possible keys.

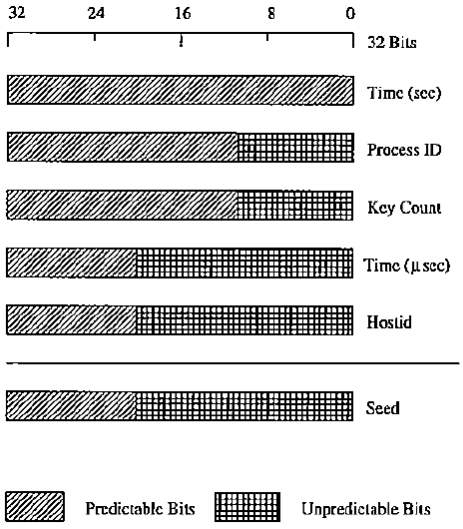

Figure 1.1: Kerberos random number generator seed (reprinted from [1]). Only lower 20 bits are unpredictable which reduces attack complexity to 2^{20}

1.2.3 Attack to Shuffling Algorithm of Online Poker

In 1999, Software Security Group from Reliable Software Technologies analyzed the published shuffling algorithm of PlanetPoker [5]. It is found that the algorithm

used by ASF Software, Inc., the company that produces the software used by most of the online poker games, including PlanetPoker Internet cardroom, suffered from many flaws. Published shuffling code is Code 1.3.

First problem found by [5] was about the shuffling of the last card in the deck. Unlike most `Pascal` functions, the function `Random(n)` actually returns a number between 0 and n-1 instead of a number between 1 and n. Hence, `random_number` in Code 1.3 is set to a value between 1 and 51. In short, the algorithm never chooses to swap the current card with the last card. When `ctr` finally reaches the last card, 52, that card is swapped with any other card except itself. Namely, this shuffling algorithm never allows the 52nd card to end up in the 52nd place. This is an obvious, but easily correctable, violation of fairness.

Reliable Software Technologies Software Security Group found the shuffling method as second problem in the algorithm [5]. In the original shuffling algorithm, each card i is swapped with a card from the range [1, n]. This causes uneven probabilities for card positions in the deck; because, number of total probabilities is n^n whereas the number of possible distributions of the deck is just $n!$. In [5], this problem is illustrated for $n = 3$ case. To solve this problem, swapping each card i with a card from the range [i, n] is suggested as solution. This change is important because the $n!$ number of distributions means that the new shuffling algorithm generates each possible deck only once. Notice that each possible shuffle is produced once and only once so that each deck has an equal probability of occurring. Now that's fair!

Last problem found by [5] is about our main topic, random number generation. Recall that in a real deck of cards, there are 52! (approximately 2^{226}) possible unique shuffles. Also recall that the seed for a 32-bit random number generator must be a 32-bit number, meaning that there are just over four billion possible seeds. Since the deck is reinitialized and the generator re-seeded before each shuffle, only four billion possible shuffles can result from this algorithm. Four billion possible shuffles is alarmingly less than 52!. Now, the worse part comes. `Pascal` function `Randomize()` chooses a seed based on the number of milliseconds since midnight thus the number of possible decks now reduces to 86,400,000—the

```
procedure TDeck.Shuffle;
var
        ctr: Byte;
        tmp: Byte;

        random_number: Byte;
begin
        { Fill the deck with unique cards }
        for ctr := 1 to 52 do
                Card[ctr] := ctr;

        { Generate a new seed based on the system clock }
        randomize;

        { Randomly rearrange each card }
        for ctr := 1 to 52 do begin
                random_number := random(51)+1;
                tmp := card[random_number];
                card[random_number] := card[ctr];
                card[ctr] := tmp;
        end;

        CurrentCard := 1;
        JustShuffled := True;
end;
```

Code 1.3: The flawed ASF shuffling algorithm (taken from [5]). The algorithm starts by initializing an array with values in order from 1 to 52, representing the 52 possible cards. Then, the program initializes a pseudorandom number generator using the system clock with a call to `Randomize()`. The actual shuffle is performed by swapping every position in the array, in turn, with a randomly chosen position. The position to swap with is chosen by calls to the pseudorandom number generator.

```
Start with fresh Deck
Get random seed
For CT = 1, While CT <= 52, Do
        X = Random number between CT and 52, inclusive
        Swap Deck[CT] with Deck[X]
```

Algorithm 1.2: Pseudocode of secure shuffling algorithm (taken from [5]). By using securely seeded random number generator, swap each card CT with a card from the range [CT, 52]. This simple card-shuffling algorithm, when paired with the right random number generator, produces decks of cards with an even distribution.

number of milliseconds in a day. 86 million is alarmingly less than 52!. In [5], the worse of the worse case was found. By synchronizing their program with the system clock on the server generating the pseudorandom number, they were able to reduce the number of possible combinations down to a number on the order of 200,000 possibilities. After that move, the system was captured, since searching through this tiny set of shuffles is trivial and can be done on a PC in real time.

As a solution to all of the problems described above, the algorithm in Algorithm 1.2 is suggested. This solution comes from the analogy between cryptographic key length (which is directly proportional to the strength of many cryptographic algorithms) and the size of the random seed that is used to produce a shuffled deck of cards.

1.2.4 Attack to Java Session-ID Generation

More recently, RNG used by Tomcat, the Apache Java Servlet, in the Java Servlets mechanism is analyzed and it has been shown how someone can exploit a flaw in the implementation of Java Servlet session-ID to impersonate another client [8]. Since HTTP is stateless, the method must be implemented to manage sessions between the client and the server. Many commercial sites use mechanisms like cookies and URL rewriting which are both based on session-ID to keep a session state at the client side. The reason why sessions should be stateful is that it makes keeping track of shopping baskets, customer preferences, previous

transactions and many other things possible. Hence session ID is important and it is the only thing represents client; i.e. an attacker can impersonate the client if he obtains session-ID of the client. In order to prevent impersonation and session stealing, the server generates a session-ID token, represented by a large random number. An impersonator should have difficulty guessing the correct token, because of the large search space. This is true only if the RNG generating that token is strong.

Gutterman and Malkhi analyzed Java Virtual Machine PRNG (Pseudorandom Number Generator), used by Tomcat servers to generate session ID tokens [8]. It uses two methods for random number generation. First method used by Tomcat servers depends on `/dev/random` and the attack is not applicable for this case. Second method is Java PRNG and it has two versions, one is `java.util.Random`, and the other is `java.security.SecureRandom`. The former is LCG (Linear Congruential Generator) while the latter is a stronger PRNG with a 160-bit state, and uses SHA-1 for transition function. Both generate random numbers recursively, starting with an initial seed. This seed has two entropy inputs which are `toString()` value of `org.apache.catalina.session.ManagerBase` and time-of-day of the server's uptime in milliseconds. If server's uptime is guessable in an accuracy of day, then it will have 2^{26} possible values. As a worst case scenario, if server's uptime is guessable in an accuracy of year, then it will have 2^{35} possible values. Other entropy input, which is `toString()` method from Java Objects Class, returns a String whose value is `getClass().getName()+"@"+Integer.toHexString(hashCode())`. Only the result of the method `hashCode()` is not fixed in the result. When examining the method `hashCode()`, Gutterman and Malkhi discovered that some implementations (e.g. the Microsoft Windows platform) use LCG. This makes the `hashCode()` value predictable. In practice, they show that this value contributes not more than 8 unpredictable bits. As a result, in order to guess and steal the session-ID, attacker needs only 2^{34} - 2^{43} guesses which is feasible on a home computer. After correctly estimating the session-ID, an attacker can impersonate the client.

```
    MD_Update(&m,buf,j);
    [ ... ]
    MD_Update(&m,buf,j); /* purify complains */
```

Code 1.4: Debian code lines before change. As it can be seen, there is a comment indicating Purify tools gives warnings at this line.

```
/*
 * Don't add uninitialised data.
        MD_Update(&m,buf,j);
*/
    [ ... ]
/*
 * Don't add uninitialised data.
        MD_Update(&m,buf,j);    /* purify complains
*/
```

Code 1.5: Debian code lines after change. Recall that the lines causing warnings by Purify are commented out.

1.2.5 Random Number Bug on Debian OpenSSL

In 2008, Luciano Bello discovered that the random number generator in Debian's OpenSSL package is predictable [9]. Cryptographic key material may be guessable because of an incorrect Debian-specific change to the OpenSSL package. The bug in question was caused by the removal of the lines which resulted in the Valgrind and Purify tools to produce warnings about the use of uninitialized data in any code that was linked to OpenSSL [10]. The initial code snippet Code 1.4 is changed to Code 1.5 in order to eliminate the warnings. Removing this code has negatively affected the seeding process for the OpenSSL PRNG. Instead of mixing in random data for the initial seed, the only *random* value that was used became the current process ID. This resulted in a very small number of seed values being used for all PRNG operations such as key generation.

1.2.6 Hacking of PlayStation 3 Root Key

In December 2010, a group of coders operating under the name Fail0verflow had managed to exploit a weakness in the PlayStation 3's encryption system, thereby gaining the root key required to run any software on the machine [11]. Sony uses digital signature to check whether the firmware and files are modified and valid. The software and files must be signed with Sony's private key in order to run on Sony PlayStation. This is not possible without knowing the private key. Sony used Elliptic Curve Digital Signature Algorithm (ECDSA) for signing purpose and ECDSA has a property such that if there are two files signed with the same key, then it is possible to extract that key. The weakness lies right here. Normally, private key is randomly generated and there is no way that someone can guess, calculate, or use a timing attack, or any other type of attack in order to find that private key. However Sony made a huge mistake in their implementation, they used the same private key everywhere, which means that if you have two signatures, both with the same key, then you can calculate the key using two signatures. After calculating the key, any software can be run on PlayStation. Choosing a constant value for private key is a huge mistake in cryptography, and concrete example of this mistake and its results are illustrated in this section.

1.2.7 Common Factors of RSA Keys

The most widely used cryptosystem for authentication purpose is RSA. The RSA cryptosystem is intended to be based on the difficulty of factoring large numbers. An RSA public key consists of a pair of integers: an encryption exponent e and a modulus N, which is a large integer that itself is the product of two large primes, p and q. If an adversary can factor this integer N back into its prime factors p and q, then the adversary can decrypt any messages encrypted using this public key. However, even using the fastest known factoring algorithm, to public knowledge nobody has yet been able to factor a 1024-bit RSA modulus.

It is vitally important to the security of the keys that they are generated using random inputs. If the inputs used to generate the keys were not random, then an

adversary may be able to guess those inputs and thus recover the keys without having to laboriously factor N. In February 2012, two groups of researchers revealed that large numbers of RSA encryption keys that are actively used on the Internet can be cracked because the random numbers used to generate these keys were not random enough [12, 13].

This problem and the reasons are discussed in Section 3.2.

1.3 Related Work

Having mentioned some flaws related to random number generator, the works on the analysis of random number generator of operating systems will be discussed in this section.

1.3.1 Analysis of Windows RNG

Windows is not an open source operating system; therefore, analysis of its functions and executables requires a lot of effort and patience. After reverse engineering part, analysis part takes a vast amount of time. Despite all the difficulties, the pseudorandom number generator used by Microsoft in Windows were analyzed in [14, 15]. `CryptGenRandom` function in Windows 2000 has been analyzed and its operation was revealed without assistance from Microsoft. As a result, it was shown that random number generation in Windows 2000 is far from being genuinely random — or even pseudorandom. These flaws exist even in Windows XP but they were solved after Windows XP SP3 by changing random number generation algorithm.

It has been found that the WRNG has a complex and layered architecture which includes entropy rekeying every 128 KBytes of output which means that WRNG does not use entropy measurements and is, therefore, not blocking. Also WRNG uses RC4 and SHA-1 as building blocks, but RC4 does not provide any forward security. Therefore, the attacker can learn future outputs in O(1) time

and compute past outputs in $O(2^{23})$ time. Given how the operating system operates the generator, this means that a single attack reveals 128 KBytes of generator output for every process. Another property of WRNG is that it runs in user mode rather than in kernel mode; hence, it is easy to access its state even without administrator privileges. The last important property of WRNG is that it keeps a different instance of the generator for every process; i.e. every process has its own random number pool.

1.3.2 Analysis of Linux RNG

The earliest analysis on Linux RNG is roughly done in [16]. In this work, related Linux RNG is described with their main components and highly shallow explanations.

The first comprehensive analysis and examination of Linux RNG is done in [4]. In this thesis, Linux kernel version 2.6.10 is studied. There did not exist a detailed description of the random number generation process before this study; the codes were analyzed statically and dynamically by simulating the code in user mode. After these works, critical flaws have been found and they are described in a more technical manner in [17]. The authors demonstrated an attack on the forward security of the generator, with an overhead of 2^{64} in most cases and an overhead of 2^{96} in other cases. Additionally, they showed that blocking the `/dev/random` device permanently is possible by reading from it excessively. Moreover they showed that Linux RNG implementation on the wireless routers may be weak because they do not have enough entropy inputs.

The problems in the wireless routers are examined deeply in [18]. Different wireless routers are investigated for entropy sources and their random number output sequences. The current thesis furthermore reviews random number generator in Linux kernel version 2.6.22 is reviewed and compares it with the generator in Linux kernel 2.4 series.

Until [19], the works on Linux RNG are only review of the code; there was not any theoretical analogy for the process. In this paper, the stages of random number generator in Linux kernel version 2.6.30.7 are theoretically and mathematically studied.

In [20], entropy estimator of Linux RNG is interpreted as polynomial interpolation. This interpretation is explained in Chapter 4.

Entropy transfers on different types of machines are studied thoroughly in [21]. It is found that the major entropy provider is disk and major consumer of the random numbers is kernel itself.

Because of the early random generation on Linux systems, same private keys are generated worldwide [12, 13]. The reason is insufficient entropy and this problem is discussed in Section 3.2.

Chapter 2

Android RNG

"Any one who consider arithmetical methods of producing random digits is, of course, in a state of sin."
—John von Neumann

In this section, we will describe general Android RNG structure, its basis on `Java.Security.SecureRandom` implementation and link with Linux kernel RNG.

Before describing Android random number generation, it will be helpful to introduce some basics of Linux RNG using the comments in [22]. Linux RNG has three output interfaces. First one is `void get_random_bytes(void *buf, int nbytes)` which is used within the kernel and this method produces random outputs for intra-kernel processes.

The other two interfaces are two character devices `/dev/random` (blocking) and and `/dev/urandom` (nonblocking). First one is suitable for cryptographic usage; it will only return a maximum of the number of bits of randomness (as estimated by the random number generator) contained in the entropy pool. If there is not enough randomness in the pool, then it will block the process until the sufficient entropy is collected in the pool. The second device does not have this

limit, and will return as many bytes as are requested. If randomness is not enough, then it will not stop producing random numbers; new outputs will be merely cryptographically strong. For many applications, however, this is acceptable.

After necessary background for Linux RNG is explained, now Android RNG can be explained. As described in [23], Android Operating System has `java.util.Random` class for random number generation purposes; this class returns pseudorandom values. However, this class is not proper to use for cryptographic purposes. There is `java.security.SecureRandom` class which extends formerly mentioned `java.util.Random` to generate cryptographically secure pseudorandom numbers.

2.1 General Structure of Android RNG

We examined the sources firstly in order to analyze random number generation process. In the source code of `SecureRandom.java`, there is no explicit algorithm defined; this class uses predefined algorithms to generate random numbers. These predefined algorithms are provided by different *Service Providers* which must extend `SecureRandomSpi.java` class. There is a default Service Provider Interface in Android source codes: `SHA1PRNG_SecureRandomImpl.java`.

In the description of `SHA1PRNG_SecureRandomImpl.java` class, it is said that generation of pseudorandom bits is performed by using the implementation technique described in *Random Number Generator (RNG) algorithms* section in Appendix A of [24] and the algorithm is named SHA1PRNG. In the description, it is claimed that SHA1PRNG implementation follows the *IEEE P1363 standard, Appendix G.7: Expansion of source bits*, and uses SHA-1 as the foundation of the PRNG. It computes the SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by one for each operation. From the 160-bit SHA-1 output, only 64 bits are used. So it can be said that SHA1PRNG is a somewhat secure and totally deterministic algorithm, i.e. although it provides

```
private static final String DEVICE_NAMES[ ] =
                { "/dev/urandom" /*, "/dev/random" */ };
```

Code 2.1: Android RNG uses only `/dev/random`. Recall that `/dev/random` is commented out.

forward security thanks to one-wayness property of SHA1 algorithm, it does not provide backward security.

The only input which makes SHA1PRNG produce different output sequences is its seed. This seed can be provided manually; however it is dangerous to seed `SecureRandom` with the current time because current time value is more predictable to an attacker than the default seed. Therefore, the default is generally used in SHA1PRNG algorithm.

Default seed is provided by `getRandomBits` method of `RandomBitsSupplier` class. In the class description it is indicated that the source for true random bits is either one of Linux's devices: `/dev/urandom` or `/dev/random`. The source for true random bits depends on which one is available; if both of them are available, then the first one is used. However, `/dev/random` is commented out in line 70 of `RandomBitsSupplier.java` as shown in Code 2.1.

Although `/dev/urandom` alone is not safe enough to use for cryptographic purposes, this may not cause any crucial problem; because outputs of `/dev/urandom` is used as seed for cryptographically secure PRNG of Java-Android. Furthermore, in embedded devices, `/dev/urandom` is generally used as the only source for random numbers [18].

The fact that even Android's secure random generator is using `/dev/urandom` arises the question of whether `/dev/random` is used in any program. As indicated in [25], JVM relies on `/dev/random` by default for UNIX platforms. However, this can potentially block some processes; because, on Linux, `/dev/random` waits for a certain amount of entropy to be collected on the host machine before returning a result. Although `/dev/random` is more secure, using `/dev/urandom` is preferable if the default JVM configuration delays on some processes [25].

After these discussions on security of `/dev/urandom`, one can wonder if there is any module in Android which uses it to generate random numbers, other than `SecureRandom.java`. IBM Application Security Research Group discovered a very interesting vulnerability in Android's DNS resolver [26], a weakness in its pseudorandom number generator (PRNG), `res_randomid()`, which makes DNS poisoning attacks feasible. DNS poisoning attacks endanger the confidentiality and integrity of the target machine. For instance, DNS poisoning can be used to steal the victim's cookies, or tamper with weak applications' update mechanisms in order to execute malicious code. After Android version 4.1.1, in order to eliminate this vulnerability, random numbers are now taken from `/dev/urandom` which should have enough entropy when the call is made [26].

At this point, it can be seen that security of Android RNG relies on the security of Linux RNG; i.e. outputs of `/dev/urandom`. If the outputs of Linux RNG can be predicted, then Android RNG would be totally deterministic. As a result, there will be no cryptographically secure random numbers that applications can use.

2.2 Linux RNG - v2.6.29

We have two options for Android kernel versions: 2.6.29 and 3.4. For version 3.4, compilation could not be completed; therefore, we study random number generator in Linux kernel version 2.6.29 in this thesis.

There are three different pools-state vectors in the random number generator system: input pool (512 bytes), blocking pool `/dev/random` (128 bytes), and nonblocking pool `/dev/urandom` (128 bytes). Entropy provided from disk events, user inputs and interrupts affect the input pool. Outputs are read from the output pools which are `/dev/random` and `/dev/urandom`. Also there are transfer events between input pool and output pools. All these structures, which are all elements and processes in Linux RNG are shown in Figure 2.1.

Now, Linux RNG can be described in parts.

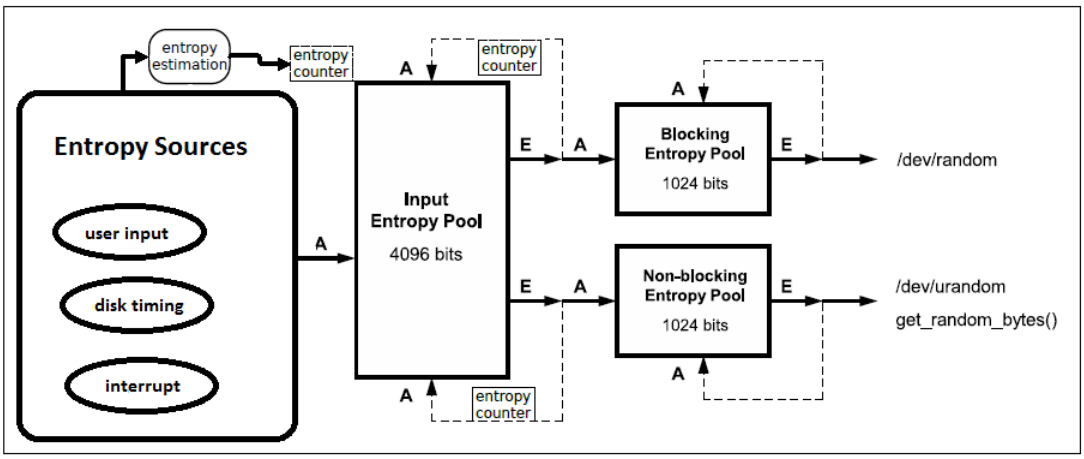

Figure 2.1: Linux RNG full scheme. Entropy is collected from the entropy sources. Then this entropy is mixed into the input pool while estimating its randomness. Random numbers are extracted from the pools using extraction algorithms. During extraction, there is also feedback portion to ensure forward security. After extracting from input pool, random numbers are mixed into the secondary pools: blocking and nonblocking pools. Requested random numbers are extracted from blocking pool or nonblocking pool with respect to requester interface.

2.2.1 Initialization

When Linux random number generator is initialized, the contents of all pools (input, blocking and nonblocking) and their entropy counts are reset to zero. Then all the pools are mixed with their individual initialization time and system constant. This procedure will be described in Section 3.2.

At the startup, Linux boot process does not provide much entropy in the different sources available to the RNG. Therefore, the designer of the Linux RNG recommends a script which generates data from `/dev/urandom` and saves it in a file, and writes the saved data to `/dev/urandom` at startup [22]. This mixes the same data into nonblocking pool and its initial entropy increases although its entropy counter is still zero. If this is not possible, for example in Live CD systems, the nonblocking random number generator should be used with caution directly after the boot process since it might not contain enough entropy. Recall that this script may solve the detected problems

in [12, 13, 18] which are described in Section 3.2. Android developers provide more randomness during initialization by applying this recommendation. `/frameworks/base/services/java/com/android/server/EntropyService.java` file in Android source is a service designed to load and periodically save randomness for the Linux kernel.

2.2.2 Entropy Collection

There are three sources of entropy: disk, interrupts, and user-input. Randomness from these inputs are collected continuously and used to mix the input pool. In this section, entropy formation and collection will be explained.

Interrupt randomness is collected via `add_interrupt_randomness` function in `random.c`. This function is called by interrupt service routines and receives the interrupt number as a parameter. The type of entropy event is calculated by adding `0x100` to the interrupt number. The resulting value and timing information is passed to the `add_timer_randomness` function, which adds the entropy to the input pool. However, there are many interrupts which come regularly to the system and these interrupts do not make the system random; i.e. they do not provide random inputs, they can be estimated easily. Therefore, each device driver can define whether its interrupts are suitable as entropy inputs, by adding the `IRQF_SAMPLE_RANDOM` flag to the corresponding handler. However, this flag has been scheduled for removal since 2009 (as stated in the `feature-removal-schedule.txt` file within the kernel source tree), due to several misuses. With kernel version 3.6, it is removed completely. For interrupt randomness, a new pool, named `fast_pool`, is defined and interrupt randomness mixes that new pool directly instead of mixing the input pool.

Disk randomness is collected via `add_disk_randomness` function in `random.c`. This function is called after completion of a disk I/O operation. The type of entropy event is calculated as in Algorithm 2.1.

```
type-value = 0x100 + ((major << 20) | minor)
```

Algorithm 2.1: Disk randomness calculation. `type_value` is dependent on only major and minor value of the related disk; therefore, the different values it can take does not exceed eight on average machines.

```
num = (type << 4) ⊕ code ⊕ (code >> 4) ⊕ value
```

Algorithm 2.2: Input randomness calculation. Although all *type*, *code*, and *value* are sixteen-bits length, unknown bits in *num* is eight for keyboard and twelve for mouse interrupts as shown in Table 2.1

The resulting value is passed to the `add_timer_randomness` function together with timing information. It can be seen that different accesses to the same disk will result in the same type of entropy event. Also assuming an average machine has no more than eight disks, the type-value actual span is limited to three bits.

Input randomness is collected via `add_input_randomness` function in `random.c`. This function is called sequential to one of the input events occurs. The type of entropy event indicates whether an event is related to a key, button, mouse or touchpad. These event codes are defined in `/usr/include/linux/input.h`. The function checks for repeating events (with same value), and avoids using them for entropy collection. The *type*, *code*, and *value* of the input events are mixed to get the type of entropy event using the algorithm in Algorithm 2.2. The result value is passed to the `add_timer_randomness` function together with timing information.

Table 2.1 taken from [17] presents the number of unknown bits for each type of event. Note that the actual entropy of these events is much lower, as most of them are predictable to a large extent. However, timing information increases the uncertainty, thereby the security.

Keyboard	Mouse	Hard-Drive	Interrupts
8	12	3	4

Table 2.1: The number of unknown bits in operating system events (taken from [4]). Actual entropy from these events is much lower, hence LRNG uses timing information to increase uncertainity.

Let e_n denote the n^{th} event and t_n denote its timing.
Define the variables below:
first level delta : $\delta_n = t_n - t_{n-1}$
second level delta : $\delta_n^2 = \delta_n - \delta_{n-1}$
third level delta: $\delta_n^3 = \delta_n^2 - \delta_{n-1}^2$
All of t_n, δ_n, δ_n^2 and δ_n^3 are 32-bit long.
Entropy added by e_n = min($\log_2$(min(δ_n, δ_n^2, δ_n^3)),11)

Algorithm 2.3: Entropy estimation algorithm. Three levels of time differences are calculated and logarithm of minimum of these values is taken as entropy estimation.

2.2.3 Entropy Estimation

After collecting entropy from the disk, user, or interrupts; `add_timer_randomness` is called. In this function, timing information and the return values of the preceding functions (`add_disk_randomness`, `add_input_randomness` and `add_interrupt_randomness`) are combined and this result is passed to `mix_pool_bytes` function which mixes the pool. Beside this task, entropy is estimated using timing information in this function.

The LRNG estimates the amount of entropy of an event as a function of its timing only, and not of the event type. The reason for choosing this calculation method and how it works will be explained in Chapter 4. The estimation of the entropy provided by the events is handled using the Algorithm 2.3.

As it can be seen from Algorithm 2.3; initially, three levels of δ (time difference) are calculated for each particular event. After this step, minimum of those three level δ is taken and logarithm of the least significant eleven bits of chosen δ is returned as entropy estimation.

2.2.4 Entropy Addition and Mixing the Pool

As it is indicated, `add_timer_randomness` function prepares the input for `mix_pool_bytes` function whose task is adding the entropy by mixing the pool. This procedure mixes one byte at a time by first extending it to a 32-bit word, then rotating it by a changing factor and finally mixing it in the pool by using a twisted generalized feedback shift register(TGFSR) [27]. For this purpose, each pool maintains a primitive polynomial. The input pool's polynomial is $x^{128} + x^{103} + x^{76} + x^{51} + x^{25} + x + 1$. The blocking and nonblocking pool have the same polynomial: $x^{32} + x^{26} + x^{20} + x^{14} + x^7 + x + 1$. There is also twist table, whose values are `0, 0x3b6e20c8, 0x76dc4190, 0x4db26158, 0xedb88320, 0xd6d6a3e8, 0x9b64c2b0, 0xa00ae278`. All of these processes are shown in Figure 2.2.

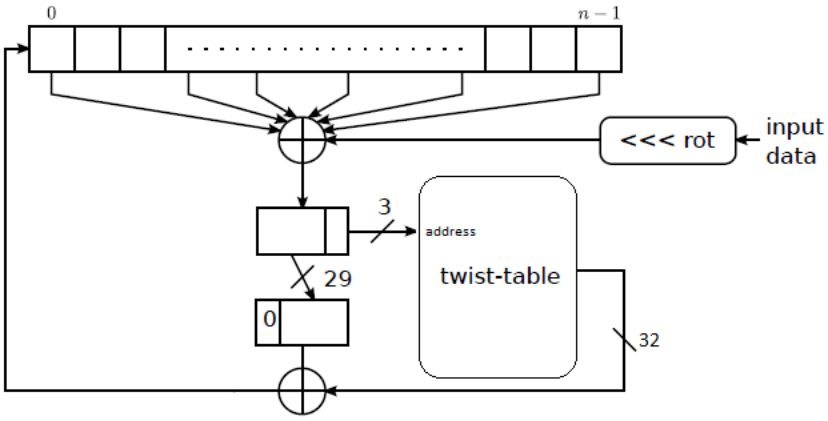

Figure 2.2: Linux RNG mixing function. Mixing function is similar to general TGFSR structure. Input data is rotated by the changing rotation value and $\oplus$'ed with the values from the pool which are taken with respect to pool's polynomial and current index. Then lowest three bits are used as indices for twist table. The value from twist table and remaining 29 bits are $\oplus$'ed and result is written into the current index.

This entropy addition process is mostly similar to TGFSR but there is a difference; new state depends not only on the previous state, but also on the input entropy word. Due to this difference, some properties of TGFSR cannot

be applied anymore; i.e. the process is no longer a linear function, and the long period cannot be guaranteed [17]. Moreover polynomial of this TGFSR is examined in [19] and it is found that the polynomial is not irreducible. Thus, the resulting TGFSR does not achieve maximum period. However, this does not cause any practical attack on the system.

In short, mixing algorithm is designed so that it can diffuse entropy into the pool and no entropy gets lost. Note that the exact details of the entropy addition algorithm do not affect the results that we show in this thesis.

2.2.5 Entropy Extraction

When random number output is needed from blocking or nonblocking pool; or when entropy transfer is needed from input pool; entropy extraction routine whose name is `extract_entropy` is called. When it is called, it returns the requested number of bits by calling `extract_buf` several times. Extraction algorithm runs under `extract_buf` function.

Extraction algorithm starts with generating a hash across the pool; 16 words (512 bits) at a time. After generating the hash to whole pool, resulting hash is mixed back into the pool in order to prevent backtracking attacks (where the attacker knows the state of the pool with the current outputs, and attempts to find previous outputs), unless the hash function can be inverted. After mixing, 16-word portion of the pool is taken from the pool and its hash is calculated (maintaining to chaining hash). In case the hash function has some recognizable output pattern, resulting hash is folded in half. In order to fold the output of hash function; first word is $\oplus$'ed with fifth word, second word is $\oplus$'ed with fourth word and first two bytes of third word is $\oplus$'ed with last two bytes of third word. The result of folded hash is output of extraction process which has 10-byte size.

Linux RNG uses SHA1 as hash function in this process. Each SHA1 block has size of 512 bits; therefore pool is hashed 16 words (512 bits) at a time. The output and internal states of SHA1 have size of 20 bytes which means that 20

byte hash value is mixed into the pool during the extraction process. Finally, hash output is folded into half and 10 byte-sized extraction output is obtained.

2.2.6 Entropy Accounting

Up to this point, the main parts of the Linux random number generator are described. In order for these processes to work properly, some auxiliary functions and variables are needed. The most important one is entropy estimation variable for each pool and accounting functions for this variable. When entropy is added to the pool, entropy estimation of that pool is increased by the entropy estimation of input. This is done in `credit_entropy_store` function. Similarly, when entropy is extracted from the pool, entropy estimation of that pool is decreased by the entropy estimation of output. This is done in `account` function. In light of this information, these auxiliary variables and functions will be examined in the same order of explanation of Linux random number generator's main functions.

When pools are initialized, entropy estimation is set to zero for each pool. Mixing the pool with time and system constant does not increase the entropy estimation of the pools.

During the entropy collection process, after collecting the events, entropy estimation of related event is calculated and then input pool is mixed with related event input. At the same time, entropy estimation of the input pool is increased by the entropy estimation of the event; i.e. the amount of the incrementation is between 0 and 11 bits. In other words, entropy estimation function works only for the input pool.

After collecting entropy in the input pool, some entropy is transferred to the secondary pools—blocking pool and nonblocking pool—when requested. This transfer process is defined in `xfer_secondary_pool` function. Subsequent to doing some necessary checks, requested amount of entropy is transferred from input pool to one of the secondary pools. For example, if 8-byte entropy is needed, 64 bits of entropy is decreased from the input pool and the same amount of entropy is

added to the requester pool. Recall that, `extract_entropy` function is called for the input pool during the extraction of entropy from the input pool and `mix_pool_bytes` function is called for the requester pool during the mixing the requested amount of entropy bits.

Similar progress is valid for outputting random number process. When a user or kernel function requests output from random number generator, output bits are extracted from the related secondary pool and its entropy is decreased by the amount of entropy given to the requester.

Similar to the initialization process, writing to `/dev/random` or `/dev/random` increases the randomness of blocking and nonblocking pools respectively. However, this does not increase entropy estimation of the related pool similar to the initialization events.

2.3 Linux RNG - After v2.6.29

There are lots of different kernel versions in [22] and many of the major versions are still under development. For example, on April 29th, 2013, kernel versions 2.6.34 series, 3.0 series, 3.2 series, 3.4 series, 3.8 series and 3.9 series were still improving and changing independently. Therefore, bigger major version number may not mean newer kernel version. In this section, initially two different emulated-Android kernel versions will be described: v2.6.29 and v3.4.0. Then changes to Linux RNG will be represented cumulatively. First and the biggest modification is realized with version 3.4.9. After this version, there is not any major modification in terms of algorithm. The newest kernel versions 3.0.75, 3.2.44, 3.4.42, 3.6.11, 3.7.10, 3.8.10 and 3.9.0 have nearly the same algorithm, there are just minor modifications which does not change the running of the main cores of the RNG algorithm.

2.3.1 Differences between v2.6.29 and v3.4.0

In version 3.4.0; if there is an architectural random number generator installed on the system, then this is used in `get_random_bytes` interface instead of using nonblocking pool as random number source. This architectural random number generator uses `RDRAND` instruction (Intel® Secure Key, previously code-named Bull Mountain Technology) to generate random numbers. If supported, this is a high bandwidth, cryptographically secure hardware random number generator as shown in Figure 2.3 taken from [2]. In order to provide the security of random number generators, it should be resistive to the attacks. From this perspective, Intel RNG crypto and classifier blocks can always be built to thwart timing and power analysis attacks [28]. Furthermore, Intel RNG is also resistive against power glitching attacks; i.e. RNG turns itself off when voltage or temperature goes out of spec, re-initializes itself when power and voltage return to spec [28]. Beside the attack protection, Intel RNG uses built-in self-tests to evaluate whether the blocks implementing the RNG are operating correctly [28].

Additionally, this hardware-based random number generator is used in `add_timer_randomness` to set cycles to any random value instead of getting its value from CPU. Recall that jiffies are still taken from CPU, because that value is used in entropy estimation process.

Another change in version 3.4.0 is that output of `extract_buf` is compared with its previous output. If they are the same, kernel panic message appears to indicate the problem.

Last but not least, if supported, hardware-based random number generator is used in initial mixing process instead of constant system value.

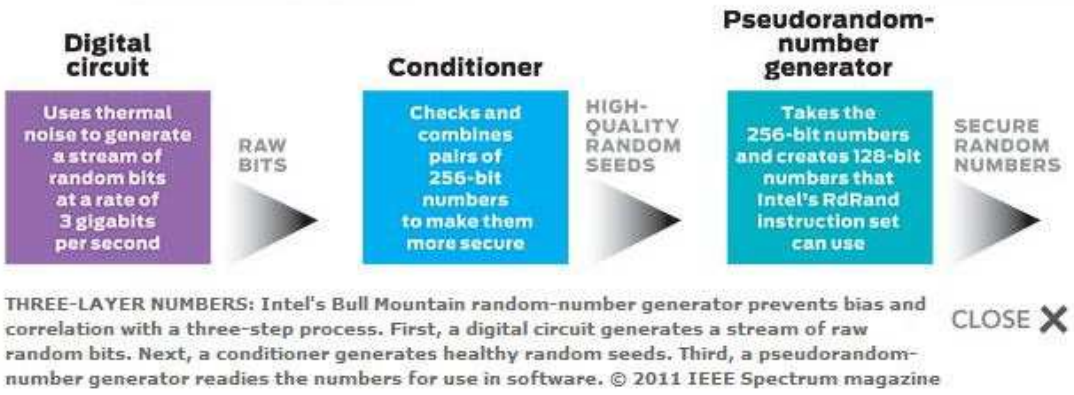

Figure 2.3: Intel's Bull Mountain random number generator (taken from [2]). Intel's new hardware RNG uses thermal noise as source, after applying three layers it outputs secure random numbers which can be requested by `RdRand` instruction.

2.3.2 Differences between v3.4.0 and v3.4.9

With version 3.4.9, hardware-based random number generator output is taken from `get_random_bytes` interface and `get_random_bytes_arch` interface is created for this purpose. Namely, `get_random_bytes` will generate software-based random numbers as before and `get_random_bytes_arch` will be used for more secure random number requests.

Another improvement in version 3.4.9 is that if supported, hardware-based RNG output is $\oplus$'ed with `extract_buf` output in order to make Linux RNG output more random. After this modification, outputs are totally not guessable even `extract_buf` function has some recognizable patterns assuming that hardware random number generator provides secure random numbers.

Furthermore, `add_device_randomness()` function is added to Linux RNG. This function provides device- or boot-specific data and mixes them into the input and nonblocking pools to help initialize them to unique values. This does not increase entropy estimation of the pools, but it initializes the pools to different values for devices that might otherwise be identical and have very little entropy available to them (particularly common in the embedded world).

Last change in this version is about adding 128-bits sized new pool to the system: `fast_pool`. This pool is designed for interrupt randomness actions. If an interrupt is received, input to `fast_pool` is formed using interrupt information, timing information and instruction pointer information. Then input is mixed into the `fast_pool` using similar algorithm shown in Figure 2.4 which is similar to the original mixing algorithm. Before this modification, it was too expensive to mix the input pool on every interrupt. Also flooding the CPU with interrupts could theoretically cause bogus floods of entropy from a somewhat externally controllable source [29]. This modification solves the problem by limiting the interrupt randomness addition to just once a second or after 128 interrupts, whichever comes first. When this limit is achieved, all content of `fast_pool` is mixed into the input pool or nonblocking pool. During initialization procedure, `fast_pool` is mixed into nonblocking pool in order to provide more secure random in a faster manner. After initialization, `fast_pool` is mixed into the input pool.

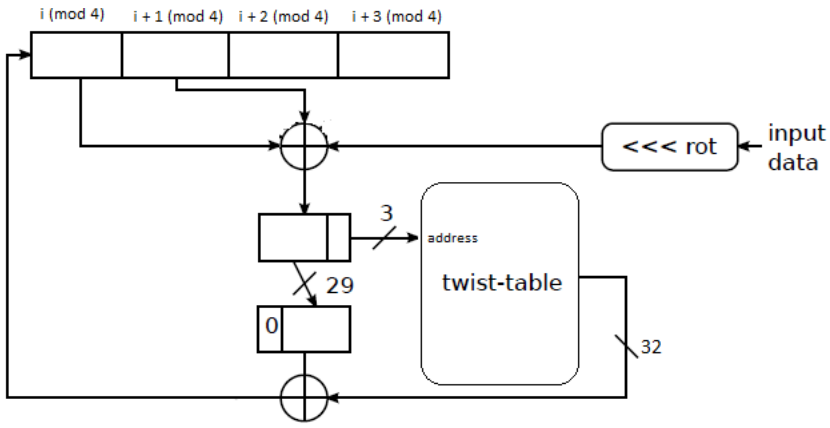

Figure 2.4: Linux RNG `fast_pool` mixing function. The algorithm is very similar to normal mixing function algorithm. Input data is rotated by the changing rotation value and $\oplus$'ed with the content of current index and next index of the pool. Then lowest three bits are used as index for twist table. The value from twist table and remaining 29 bits are $\oplus$'ed and result is written into the current index.

2.3.3 Hardware RNG and Linux

As it is explained in the previous section, newer versions of Linux can use installed hardware based RNG as an assistant to its random number generator. According to the related documentation in the kernel source files (`/Documentation/hw_random.txt`), when hardware RNG is available, the device `/dev/hw_random` is created automatically. The interfaces for reading the random numbers from this device is provided by the software developed by hardware producers. These producers for each kernel version can be seen in `/drivers/char/hw_random` directory of the related source code. Newer kernel versions provide more variety of hardware RNG. There are interfaces for supported models of Intel, AMD, Niagara2, VIA, OMAP, PA Semi in kernel version 2.6.29. In the latest kernel version 3.9; the interfaces for supported models of Atmel, Broadcom, Octeon, Free-Scale, PicoChip, PowerPC and Exynos are added. Using these interfaces, reading from hardware RNG device `/dev/hw_random` is simply done by using `read()` command. Information about installed hardware RNG on the system can be seen by checking `rng_available` and `rng_current` attributes in `/sys/class/misc/hw_random` node. The former attribute lists available hardware-specific drivers and the latter lists the one which is currently connected to `/dev/hw_random`. If the system has more than one RNG available, it is possible to change the one used by writing a name from the list in `rng_available` into `rng_current`.

A key advantage of using hardware RNG is performance. Because sampling an entropy source is typically slow since it often involves device I/O of some type and often additional waiting for a real-time sampling event to transpire. In contrast, hardware RNG computations are fast since they are processor-based and avoid I/O and entropy source delays. According to [3], since the implementation of Linux RNG is typically in software, it may be vulnerable to a broad class of software attacks; i.e. memory-based attacks or timing attacks. Moreover, the approach does not solve the problem of what entropy source to use. Without an external source of some type, entropy quality is likely to be poor. For example, sampling user events (e.g., mouse, keyboard) may be impossible if the system

resides in a large data center. By asserting these reasons, Digital Random Number Generator (DRNG) usage is promoted in [3]. Cascade construction RNG model is used in the DRNG; i.e. processor resident entropy source is used to repeatedly seed a hardware-implemented cryptographically secure PRNG. This structure is shown in Figure 2.5 taken from [3]. Recall that Figure 2.5 is more technical version of Figure 2.3. Furthermore, it represents a self-contained hardware module that is isolated from software attacks on its internal state [3]. As a result, Intel advocates that DRNG is a solution that achieves RNG objectives with considerable robustness: statistical quality (independence, uniform distribution), highly unpredictable random number sequences, high performance, and protection against attack.

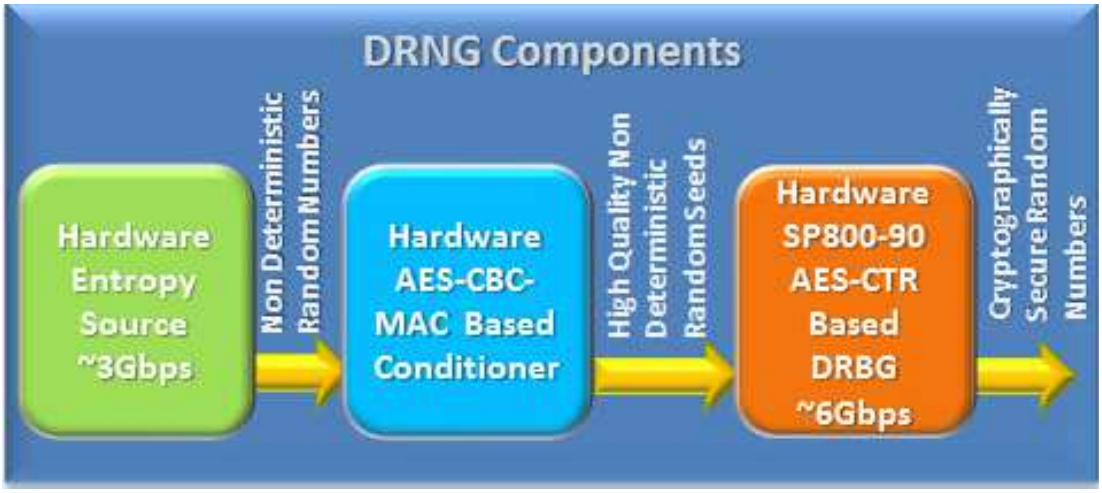

Figure 2.5: Digital random number generator's cascaded component architecture (taken from [3]). Intel's new hardware RNG uses hardware-based components in all of the three layers.

In order to check the performance, security and reliability of the RNG, Intel asked Cryptography Research to review the design of the RNG [30]. DRNG uses post-processing to obtain secure seeds from the entropy source. However, in the analysis report [31], it is found that defects in the entropy source become more difficult to observe due to using post-processing. Therefore, users of the RNG may experience a more difficult time assessing the quality of the underlying entropy source, and some catastrophic failure modes can actually become hard

to detect [31]. As a solution, the Intel Ivy Bridge designers incorporated a circuit to monitor the health of the entropy source. In addition, while raw access to entropy source output is not available on production parts, test parts can provide direct access to entropy source outputs to make analysis easier. As a whole, the Ivy Bridge RNG is a robust design with a large margin of safety that ensures good random data is generated even if the entropy source is not operating as well as predicted and in all cases, users should check the carry flag after each call to the RNG to verify that it is working properly and the random data received is valid [31].

There are many processors which support hardware RNG as indicated above. However, many processors used actively nowadays do not have hardware RNG installed on them. Hardware RNG is used when the security requirements are hard to satisfy. Recall that, the cost of adding hardware RNG to the system is also considerable; therefore, many producers do not add it to most of their systems.

Chapter 3

Potential Vulnerabilities During Initialization

"God doesn't play dice with the universe."
—Albert Einstein

Up to this point, all we do is merely looking at the source codes and analyzing the related works. In order to understand and analyze the system better, observing the inputs and intermediate values can be very helpful. Therefore, we modify the related kernel files to obtain some information about the running system, especially random number generator part. There are different approaches to do this. First one is to write intermediate states into the file in kernel space; however as indicated in [32], writing to file in kernel space is not recommended. Instead, writing a driver which takes desired values from kernel space and pushes them into the user space, and writing a user-mode program which interprets those values and writes them into a file is a preferable method. However, this method is also not feasible for us; because modification in Android source is necessary and the source download is approximately 8.5 GB in size; additionally we will need over 30 GB free space to complete a single build, and up to 100 GB (or more) free space for a full set of builds [33].

We use kernel print function `printk` to attain desired information. This approach requires extra system calls in kernel which cause delay on the running system; yet these negative results do not effect random number generator's algorithms, internal states and running. Therefore, we prefer this approach so selected results can be printed to kernel logs. Then we obtain these values by reading the kernel logs of the emulated Android system on the computer.

Another possible method can be sending necessary packets over network as preferred in [21]. Vuillemin et al. advocate their method by saying that using network interface may have less influence over the system, because it does not cause disk event while trying to write extra information to the kernel log file. Vuillemin et. al. had to chose whether `include/linux/net.h` or `include/linux/netpoll.h` kernel API to send UDP packets. The first one is pretty similar to the standard user space socket API. The second one is more low-level and rarely used. They chose the second one, because it works even in IRQ contexts, which is required to instrument input events.

3.1 System Setup

Most of the work is done on our personal computer. It has Windows 7 as main operating system, 8 GB RAM and eight-core processor. For compiling kernel, we create virtual machine which has 3 GB RAM and quad-core processor by using Oracle VirtualBox, and install Linux-Ubuntu 12.04 on it. To emulate Android, we download Android SDK to windows, create new Android device by using AVD (Android Virtual Device) Manager and run the emulator with Android 4.1.2.

Creating the virtual machine by using VirtualBox is a simple task. After this step, we download Ubuntu 12.04 and install Ubuntu to virtual machine. Then by following instructions in [33] and its sub-pages, we download Android source code and Android kernel source code. These code lines are included in Code B.1 and Code B.2.

We download the sources and then modify related parts of the kernel. As it was mentioned before, we use `printk` to print necessary information to kernel logs. Normally, timestamps of `printk` outputs are not shown in the kernel log. In order to include these outputs, we need to change `kernel/printk.c` file by removing `if (printk_time)` condition as described in Code B.3. After this modification, all `printk` outputs will contain timestamps in kernel logs. Beside this little modification, it is necessary to increase kernel buffer size. Default kernel buffer size is 64 KB and this buffer is overwritten if log size exceed this size. Normally, all logs can be fetched using Android SDK interface functions to the file on the computer continuously; however, initial access to kernel logs takes some time. This may cause a problem, because initial logs disappear when the first access occurs. This problem is solved by changing `CONFIG_LOG_BUF_SHIFT` parameter from 16 to 17 in `/arch/arm/configs/goldfish_armv7_defconfig` file as described in Code B.4. After this modification, kernel buffer size becomes 128 KB and it is enough space to hold kernel logs until initial access occurs.

After these pre-modifications, we modify `random.c` file in `/drivers/char` directory to observe internal states of the random number generator. Then it is time to compile modified Android kernel for emulator. Again, this task is simple by following the instructions in the related page of [33] and these steps are also included in Code B.5.

Sequential to successful compilation, we copy `zImage` file from the directory of `/arch/arm/boot/zImage` to Windows environment where Android emulator is installed. Then we run the emulator with modified kernel by following the instructions in Code B.6 and get continuous kernel logs from that device by using the instructions in Code B.7 while the emulator is running.

Before starting the analysis, we check whether hardware RNG is installed on the emulator system by following the instructions in Code B.8 while the emulator is running. As it is expected, there is no hardware RNG installed on the system. Beside the emulator test, we check Samsung Galaxy Nexus and Asus Nexus 7 for hardware RNG but they have no hardware RNG, either. In the future, when the security requirements become more sophisticated and cost of adding hardware

```
prng.seed(seed)
p = prng.generate_random_prime()
prng.add_randomness(bits)
q = prng.generate_random_prime()
N = p*q
```

Algorithm 3.1: OpenSSL RSA key generation algorithm. If there is not enough entropy before seeding, seed will be the same for different startup sessions, thus same p will be generated.

RNG diminishes; it will be possible to see many Android devices with hardware RNG.

After defining complete working system, collected data is ready for analysis.

3.2 Analysis of Initialization

The weakest states of the Linux RNG is in initialization phase of the system. There is not much input to make pools random so there is not enough entropy to produce cryptographically secure random numbers. Additionally, there are not many alternative initialization sequences of the system; therefore, internal states of random number generators may be similar between different initialization procedures. As it was mentioned earlier, wireless routers have not enough entropy inputs; they produce same output stream even after system is completely started [18]. Different devices of the same manufacturer may also give similar output streams. In [12, 13], it is shown that there are many common RSA private keys on the Internet. Main source of the problems mentioned here is producing the key during initialization process of the devices. There is not enough entropy, there is not any input which makes internal states of the pool different than another initialization sequences. This problem, RSA key generation trouble, can be summarized as follows. In order to see the problem clearly, it will be helpful to see OpenSSL RSA key generation algorithm in Algorithm 3.1:

Now suppose poor entropy at startup. Then same p will be generated by multiple devices, but different q because entropy increased differently. If N_1 and N_2 which are RSA keys from different devices are examined, it can be seen that $\gcd(N_1, N_2) = p$. This reveals private key of these systems. In [12, 13], many public keys are collected from the Internet and these public keys are searched for common prime factors. As a result, 0.4% of public HTTPS keys are factored; therefore, it is important to make sure that random number generator is properly seeded during key generation. In [34], it is indicated that OpenSSL RNG is competent, problem is in its seeding. In order to eliminate this problem, it is suggested that random number generators must be seeded by external sources properly and the characteristics of the seed source has very critical effects on random number generation.

In our experiments, initialization process is considered as the interval between power on time and the end time of disk checking process; i.e. after this point device is ready to use. The reason why we do not include the time after disk check is completed is that user-mode inputs make random number generator less guessable, thereby cryptographically secure. When user starts to use Android device, his/her operations are considered as user-input like keyboard and mouse movements; i.e. event codes of the operations (touching, tapping etc.) are defined in `/usr/include/linux/input.h`.

During the initialization process, there is not any input from the user; only entropy source to input pool comes from disk randomness. Disk-randomness inputs mostly come during disk-check process. Only two inputs come when the pools are initialized, which makes internal state of the pools different. These inputs are system time in nanosecond scale and system constant (`utsname`, which contains information about the system and device). System constant does not provide security, because it is always the same and known to attacker; i.e. any other device with same model has the same system constant. As it is recommended in [18], making system constant different for each device may improve the security of randomness. As a result, only input which affect the randomness of the input pool is system time at pool initialization. Because it has nanosecond scale, pool initialization time will not be the same for different initializations with very

/net/core/neighbour.c
/net/netfilter/nfnetlink_log.c
/net/ipv4/syncookies.c
/net/bridge/br_fdb.c
/kernel/panic.c
/lib/random32.c
/fs/binfmt_elf.c
/net/ipv4/route.c
/net/ipv4/af_inet.c
/net/core/request_sock.c
/net/netlink/af_netlink.c
/net/netfilter/nf_conntrack_core.c

Table 3.1: Random number requesting files during initialization. These files request random numbers to use in their service during initialization. These services are not as critical as key generation, so it is acceptable for these services to request random numbers with lower entropy.

high probability. In short, pools have enough randomness for simple tasks after initialization but it is still vulnerable to strong cryptographic attacks.

When we sift through the kernel logs, we see that classes request random numbers from `/dev/urandom` in Table 3.1. When we look these files and random number related parts, we do not see crucial points for security. So system may not be under risk even if the random numbers are not cryptographically strong.

After initialization, input pool has zero bits of entropy and it stays the same until first disk randomness comes as input. Initialization times of input pool change between 470 – 520 ms; first disk randomness times change between 620 ms – 670 ms. After first disk randomness, there will not be any other disk randomness until disk check part begins at the last part of initialization; i.e. about 50 – 60 seconds after power-on. During this time period, input pool stays the same. After this steady time period, disk check period comes and during that period entropy of input pool increase and when it becomes greater than 192 bits, then some bits will be transferred to nonblocking pool from the input pool.

In our simulations, initialization time can be in 50 ms interval which is 50×10^6 ns. Similarly, first disk randomness time can be in 50 ms interval. However, the

time between initialization time and first disk randomness can be from 125 ms to 150 ms. This is 25 ms interval which is 25×10^6 ns. As a result, there are total 1.25×10^{15} possibilities which is approximately equivalent to 52 bits. Therefore, it can be assumed that, after first disk randomness there are 2^{52} possible input pool states which is large enough to prevent simulating all possible states for Android devices. Namely, it can be said that it is hard to implement simulating possible states attack in [18].

Regarding the nonblocking pool, the initialization process is more complicated. Normally, initialization procedure is simple and it should be represented as explained in the following lines.

The outputs of nonblocking pool are totally deterministic if its internal state just after the initialization is known. Because there is only extracting process which does not add any randomness to pool, it just mixes the pool with its output to provide forward security. Only chance to add randomness to nonblocking pool is to transfer some bits from the input pool. However, input pool does not provide any bits until its entropy exceeds 192 bits. As it was indicated in the previous chapter, this entropy is reserved for blocking pool to prevent requester process be blocked. In our simulations, input pool entropy passes 192 bit level at the near-end of the initialization part. Hence most of the random numbers are still deterministic assuming that internal state of nonblocking pool just after the initialization is known.

However, the initialization procedure of nonblocking pool in Android devices is not so simple. As mentioned in the previous chapter, Android developers provide more randomness during initialization by writing to `/dev/urandom` file. The `/frameworks/base/services/java/com/android/server/EntropyService.java` file in Android source is a service designed to load and periodically (in every three hours) save randomness for the Linux kernel. This service carries the entropy pool information across shutdowns and startups; therefore, entropy pools is not in a fairly predictable state anymore; i.e. they will not return predictable data. As a future plan, this service will be changed in a way such that it will write entropy data at shutdown time instead of periodically. The relevant code snippet

```
public EntropyService()
{
    this(getSystemDir() + "/entropy.dat" , "/dev/urandom");
}
```

Code 3.1: The code for getting entropy file and random device names - part of `EntropyService.java`.

which shows entropy file and random device is in Code 3.1. Recall that Android uses only `/dev/urandom` again; i.e. `/dev/random` is not used. However, as it will be explained, writing to `/dev/urandom` results in mixing the same data into `/dev/random`.

In order to make this service work as expected, the data written to the pool must be unpredictable. During the initialization, `entropy.dat` file, which contains random data from previous session is mixed into `/dev/urandom` at the beginning. After this, device-specific data is mixed into `/dev/urandom`, because making the data unique to the device complicates the attack to randomness. As indicated in `EntropyService.java`, even sending non-random information to `/dev/urandom` is useful because, while it does not increase the *quality* of the entropy pool, it mixes more bits into the pool, which results in a higher degree of uncertainty in the generated randomness. Like nature, writes to the random device can only cause the quality of the entropy in the kernel to stay the same or increase. For maximum effect, information writing to `/dev/urandom` varies on a per-device basis, and is not easily observable to an attacker. As shown in Code 3.2, beside some constant data, device specific data such as serial number and variable data for instance time, carrier, baseband etc. is written to `/dev/random` to increase randomness. Note that in Android 4.2 JellyBean , the name of `EntropyService.java` is changed to `EntropyMixer.java`.

When any data is written on either `/dev/random` or `/dev/urandom` device, `random_write` function is called to perform the operation. In the implementation of this function, the same data is written to both devices by calling `write_pool` function for both devices. Therefore writing to either `/dev/random` or `/dev/urandom` device causes writing the same data to both devices. However, this

```
out.println("Copyright (C) 2009 The Android Open Source
   Project");
out.println("All Your Randomness Are Belong To Us");
out.println(START_TIME);
out.println(START_NANOTIME);
out.println(SystemProperties.get("ro.serialno"));
out.println(SystemProperties.get("ro.bootmode"));
out.println(SystemProperties.get("ro.baseband"));
out.println(SystemProperties.get("ro.carrier"));
out.println(SystemProperties.get("ro.bootloader"));
out.println(SystemProperties.get("ro.hardware"));
out.println(SystemProperties.get("ro.revision"));
out.println(new Object().hashCode());
out.println(System.currentTimeMillis());
out.println(System.nanoTime());
```

Code 3.2: The code for writing device specific information to random device - part of `EntropyService.java`. By doing this, initialization will not be the same for different devices.

```
loadInitialEntropy();
addDeviceSpecificEntropy();
writeEntropy();
scheduleEntropyWriter();
```

Code 3.3: Initialization service of Android - part of `EntropyService.java`. Firstly load the random file from the previous session, then write device specific information to random device, subsequently write new random file for next session and finally schedule the last job to do periodically.

writing operations does not increase entropy of the pools, it just mixes the data to be written into the pool. After this procedure, first periodic `/entropy.dat` file generation for next startup session is accomplished. The next period comes after 3 hours as indicated in `EntropyService.java`. The summary of these steps in `EntropyService.java` is shown is Code 3.3. As it can be seen, first the randomness from file is written to nonblocking pool, then device specific entropy is mixed into nonblocking pool. After satisfying randomness requirements, generating randomness file is executed by this service.

While scanning the resulting kernel logs, we are able to see all of the operations expressed above. `loadInitialEntropy()` function writes the data in

`/entropy.dat` to `/dev/urandom`. In this operation, `random_write` function is called once and `random_write` function calls `write_pool` function 64 times for blocking pool and 64 times for nonblocking pool. In each call of `write_pool` function, 64 bytes of data is mixed to the pool. In short, $64 \times 64 = 4096$ bytes of data is mixed into both of the secondary pools.

Then we are able to observe operation of `addDeviceSpecificEntropy()` function. During the addition of data in Code 3.2, `random_write` function is called once and `random_write` function calls `write_pool` function three times for blocking pool and three times for nonblocking pool. Total 64 + 64 + 60 = 188 bytes of data is mixed to both pools. One sample of device-specific data taken by Hex Workshop v6.6 is shown in Figure 3.1.

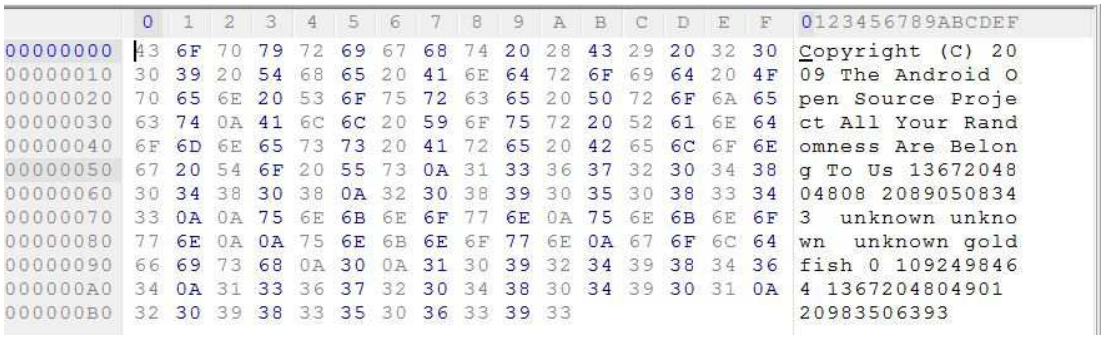

Figure 3.1: Sample output of device specific data. It is possible to link the items in Code 3.2 with this Hex Workshop v6.6 screen.

After the addition of device specific information, current entropy is written to `/entropy.dat` file to use on next startup. During this operation, `urandom_read` function is called and 4096 bytes of data is extracted from nonblocking pool. All of the extracted data is written into `/entropy.dat` file. This file should be constant until three hours from last writing process. However, we did not test if `/entropy.dat` is updated periodically. Instead of observing the periodic update, we observed that the data written to `/entropy.dat` on a session is the same as the data read from `/entropy.dat` on next session.

As a result, it is not possible to implement simple attack on nonblocking pool in Android devices. It is very hard to simulate its states, hence the attack in [18] cannot be applied to Android devices.

Chapter 4

Evaluation of Linux Entropy Estimator

"The total entropy of any isolated thermodynamic system tends to increase over time, approaching a maximum value."

—The second law of thermodynamics

Entropy estimator is crucial part of the Linux random number generator. All randomness assumptions are made based on this estimator. Therefore it is very important to analyze its algorithm and check its correctness; i.e. whether it overestimates or underestimates the entropy. Normally, it is expected of an estimator to underestimate the entropy, so random number generator should guarantee that it can provide at least estimated amount of random bits; i.e. getting 128-bit key which has 100 bit randomness case should not be possible.

As it was described in the previous chapter, Linux uses time difference to estimate the randomness of the inputs. There is not detailed information about

```
- Consider the three interpolating polynomials based upon
   the last three events.
- Compute the three interpolation errors according to the
   new event.
- Take the minimum of these errors.
- Compute the logarithm in base 2 of this minimum (bounded
   by 0 and 11).
```

Algorithm 4.1: Entropy estimation algorithm using interpolation. Entropy estimator in Linux is mathematically equivalent to the steps above.

why this particular estimator and parameters are used in the Linux source commentaries. It is a simple and cheap entropy estimator [21]. It was chosen for its cost, not for its accuracy [19]. As indicated in [19], entropy estimation is based on a few reasonable assumptions. First, it is assumed that most of the entropy of the input samples is contained in their timings. Timing contains both the cycle and jiffies counts; however, the jiffies count has a much coarser granularity. The Linux RNG uses the pessimistic estimator by basing its entropy estimation on the jiffies count only. Adding additional values, even if they are completely known, can only increase the entropy; i.e. it cannot decrease the uncertainty of the already collected data. The other assumption is that the input samples coming from different sources are independent. Hence, entropy can be estimated separately for each source which are user input, interrupts, and disk I/O and summed up in the end. The estimator keeps track of the jiffies count of each source separately. The entropy is estimated from the jiffies' difference between two events. Still, it is pretty good at detecting regularities [21]. A study [20] proposes an interpretation based on Newton polynomial interpolation. This study summarizes the estimation process as algorithm in Algorithm 4.1.

Additionally, another crucial point has to be mentioned. While running the emulator and scanning the kernel logs, we see that most of the events add zero entropy to the input pool. The reason why this occurs lies behind the implementation of the estimator. Because of this property, Linux random number generator can be characterized as conservative [17]. This may be quite a severe bottleneck for the blocking interface to the LRNG.

It is argued that `/dev/random` may fail to provide information-theoretic security even if the entropy estimator is correct [6]. For example, in Linux kernel v2.4, both streams (`/dev/random` and `/dev/urandom`) use the same entropy pool (there is only one secondary pool in Linux kernel v2.4), so the output of `/dev/urandom` leaks information also about the state of `/dev/random`. And even when this two streams use syntactically distinct pools (as in the Linux kernel v2.6), it is advocated that there is no guarantee of information theoretic security for `/dev/random` as long as they are refreshed from possibly dependent data [6].

After explaining the main logic behind the Linux entropy estimator; now it is time to see how pessimistic it calculates the entropy by comparing with two different entropy estimation methods: Maximum likelihood entropy estimator and compression.

4.1 Comparison of Linux Entropy Estimator with Maximum Likelihood Entropy Estimator

As it was indicated before, entropy estimator estimates the entropy by using only time differences between subsequent events. During this comparison, the entropy will also be calculated using only time differences. In [35], Shannon defined entropy as the quantity which will measure, in some sense, how much information is produced by the source, or better, at what rate information is produced by the source when all possible outcomes are given with their probabilities. In other words, entropy gives an idea about the randomness and uncertainty of the source.

In Linux RNG, entropy events increase the entropy of the input pool with respect to related time difference value. Namely, time difference values determine the entropy—randomness and uncertainty—of the input pool. From the similar perspective, time difference values of these events can be thought as the source and each individual event can be taken as an outcome of the source. The source

feeds randomness to input pool by giving its outcomes. If all outcomes and their probabilities are known exactly, then the entropy of the source—time difference values—could be calculated by using Shannon's entropy function. Shannon's entropy function is defined in [35] as follows:

Suppose we have a set of possible outcomes whose probabilities of occurrences are $\boldsymbol{p} = (p_1, p_2, \ldots, p_m)$. Then the entropy of the source is defined as

$$H(\boldsymbol{p}) = -\sum_{i=1}^{m} p_i \log_2(p_i) \tag{4.1}$$

In our case, we do not have exact and known values for outcomes and their occurrence probabilities. Therefore, a new question arises: Given the sample output of the source, how can we estimate the entropy of the source? The answer is stated in [36] as follows:

All output samples are assumed to be independent and identically distributed. When each symbol is included for $\boldsymbol{n} = (n_1, n_2, \ldots, n_m)$ in the observed sample set, entropy is generally estimated by substituting $\boldsymbol{p}$ in Equation (4.1) with maximum likelihood estimates of occurring probabilities of source symbols, $\hat{\boldsymbol{p}} = \boldsymbol{n}/N$, as

$$\hat{H}_{\mathrm{MLE}}(\boldsymbol{n}) = H(\boldsymbol{n}/N) = H(\hat{\boldsymbol{p}}) \tag{4.2}$$

In [36], estimator $\hat{H}_{\mathrm{MLE}}(\boldsymbol{n})$ is referred as the maximum likelihood entropy estimator.

Considering the information above, we need to customize the Equation (4.2) to our needs. In order to calculate the maximum likelihood entropy of time differences using Shannon's entropy function, we should take time differences as samples and find their frequencies.

Let N be the size of the sample outcome $\delta_1, \delta_2, \ldots, \delta_N$. Let $\hat{p}_\eta = \#\{i : \delta_i = \eta\}/N$ be the empirical frequency of η in the given sequence. Suppose there are D different outcome with frequency space $\hat{\boldsymbol{p}} = (\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_D)$. We then compute

Maximum-Likelihood Entropy of first level δ	**Maximum-Likelihood Entropy of min. δ**	**Entropy calculated by Linux Estimator**
1.5988	1.5620	0.2384
1.6775	1.6356	0.2588
1.7104	1.6596	0.3223
2.0644	2.0647	0.5053
2.3312	2.2745	0.6187

Table 4.1: Comparison of maximum likelihood entropy estimator and Linux estimator. First column shows the ML entropy estimation of first level δ, second column shows the ML entropy estimation of minimum δ and last column shows entropy estimation of Linux estimator. Recall that all estimations are based on δ values—time-differences.

Equation (4.3) to find maximum likelihood entropy estimation over the empirical data:

$$\hat{H} = -\sum_{\eta=0}^{D} \hat{p}(\eta) \log_2(\hat{p}(\eta)) \tag{4.3}$$

The result of Equation (4.3) is the maximum-likelihood entropy calculated by time differences; in other words, this is the entropy-per-sample. In estimation algorithm, there are three levels of δ and Linux estimator uses minimum of them. Therefore, we take the minimum of these δ levels for each sample like the estimator does. However, in [19] only first level δ is used for maximum-likelihood entropy calculation. Therefore, we include both calculations; i.e. we use both minimum δ and first level δ. In order to add more meaning to out comparison, we should also include per-event entropy calculated by Linux estimator. For this, we sum up the entropy added to the system for each event and divide it to the number of events. As a result, we obtain entropy-per-event calculated by Linux estimator. Now we have entropy-per-sample calculated using Equation (4.3) and entropy-per-event calculated using the Linux estimator. The comparison of these are shown in Table 4.1.

Recall that, there is not significant difference between maximum-likelihood entropy calculated using minimum δ and first level δ. However, as expected,

Linux estimator is very pessimistic on this calculation. As it was mentioned before, being pessimistic on entropy estimation is acceptable and must condition for security so Linux passes this test successfully. Also note that, last two rows of Table 4.1 have greater entropy values than the others; because, last two simulations of the system contain user-mode operation while first three simulations of the system last until the end of the disk-check process; i.e. no user-input included. This is consistent with the statement that the most vulnerable states of Android devices are during the initialization; after user starts to use the device, randomness and entropy increase dramatically.

4.2 Comparison of Linux Entropy Estimator with Compression Results

In the previous section, we compare the estimator's performance by calculating maximum-likelihood entropy of time differences by using Shannon's entropy function. In this section, we try to calculate the entropy of the input pool by measuring the randomness of the inputs which mix the input pool. These inputs are added to the input pool by mixing it with the algorithm described before. Therefore, randomness of these inputs determines the entropy of the input pool. Linux estimator use only time differences, jiffies part of these inputs as it was told before.

Entropy effectively bounds the performance of the strongest lossless (or nearly lossless) compression possible; because compression algorithms use the correlation of subsequences in its input. Random sequence has little or no correlation between its subsequences; hence, random sequences are compressed a little or they cannot be compressed. The performance of existing data compression algorithms is often used as a rough estimate of the entropy of a data block [37].

By using this information, we concatenate all the inputs to the input pool and compress the concatenated result by using the *best compression* algorithm of WinRAR v4.0.1. The compressed size can be roughly thought as the entropy

of all inputs. Input pool has zero entropy at the beginning, initialization does not increase entropy. Therefore, we include only disk-randomness inputs during initialization part to calculate and compare entropy. Results for 72 different runs of the system are shown in Table 4.2.

No	Size(byte)	Comp. Size (byte)	LRNG Est. (bit)	Ratio of Est./Comp.
1	10082	577	144	0.03
2	9026	567	139	0.03
3	8642	593	202	0.04
4	8834	573	162	0.03
5	8546	566	254	0.05
6	8834	584	166	0.03
7	8450	550	144	0.03
8	7970	560	157	0.03
9	7682	536	133	0.03
10	9266	579	156	0.03
11	9218	567	172	0.03
12	8450	573	186	0.04
13	9410	601	218	0.04
14	8882	566	134	0.02
15	7874	562	168	0.03
16	8834	576	167	0.03
17	10658	624	235	0.04
18	8258	572	170	0.03
19	8498	582	191	0.04
20	8498	577	196	0.04
21	9986	604	247	0.05
22	10274	623	251	0.05
23	9410	599	212	0.04
24	8438	604	234	0.04
25	7846	655	231	0.04

Continued on next page

Table 4.2 – *Continued from previous page*

No	Size(byte)	Comp. Size (byte)	LRNG Est. (bit)	Ratio of Est./Comp.
26	10102	679	246	0.04
27	13194	705	284	0.05
28	8930	595	187	0.03
29	10966	700	241	0.04
30	10658	636	231	0.04
31	9986	600	211	0.04
32	9398	599	191	0.03
33	11894	647	244	0.04
34	8546	584	211	0.04
35	13194	716	292	0.05
36	11370	710	252	0.04
37	8402	577	190	0.04
38	10706	628	204	0.04
39	8930	604	204	0.04
40	11542	691	257	0.04
41	8450	577	205	0.04
42	10274	613	231	0.04
43	8450	583	212	0.04
44	8258	582	205	0.04
45	8258	583	175	0.03
46	7682	544	154	0.03
47	9026	589	186	0.03
48	8978	588	199	0.04
49	9354	705	242	0.04
50	9374	584	165	0.03
51	10178	621	229	0.04
52	13154	637	238	0.04
53	12434	632	231	0.04
54	11234	626	201	0.04

Continued on next page

Table 4.2 – *Continued from previous page*

No	**Size(byte)**	**Comp. Size (byte)**	**LRNG Est. (bit)**	**Ratio of Est./Comp.**
55	10390	671	208	0.03
56	11138	643	243	0.04
57	9842	601	205	0.04
58	12714	729	270	0.04
59	9890	613	244	0.04
60	13406	655	270	0.05
61	6734	524	143	0.03
62	9334	669	218	0.04
63	8738	586	186	0.03
64	13154	628	234	0.04
65	13154	634	236	0.04
66	8630	564	138	0.03
67	8642	576	145	0.03
68	8546	582	204	0.04
69	9122	587	200	0.04
70	7682	564	205	0.04
71	10082	613	241	0.04
72	3566	348	72	0.02

Table 4.2: Comparison of WinRAR's *best compression* and Linux entropy estimator. First column shows the total size of the concatenated size of all entropy events which mix into the input pool. Second column shows compressed (with WinRAR's *best compression*) size. Third column is the entropy estimation of Linux. Last column is the ratio between Linux entropy estimation and compressed sizes which can be thought as entropy of all events. On average, entropy estimated by compression is 23 times greater than entropy estimated by Linux.

As it can be seen from Table 4.2; although concatenated inputs are compressed to 0.04-0.07 of their original sizes, it is still far, far greater than estimator's estimation. On average, entropy estimated by compression is 23 times greater than entropy estimated by Linux. In short, estimator again estimates pessimistically as it is expected in this test.

In addition to the concatenated inputs test, we test if input pools' parallel states are correlated or not. All pools are initialized to zero at the beginning. At first, time is mixed into the input pool, we can call this the first state. For all different initializations, in order to indicate first states of all input pools, we use first parallel state. All the first states of the input pools in different initialization are concatenated and compressed, which we call the compression of concatenation of first parallel states. The size of input pool is 512 bytes and the size of time input is 8 bytes. Therefore, input pool still contains lots of zeros, unchanged region after the first mixing. After mixing with time, input pool is mixed with the system information, `utsname` structure, which is the size of 390 bytes. After this second mixing process, all parts of the input pool is mixed. We call this the second state. As a third step, disk randomness is mixed, which gives us the third state. On these terms, parallel states can be defined as follows: after the first mix of the pool in all runs, after the second mix of the pool in all runs etc. Second and third states and their compressions are defined in a similar way to the first state. After collecting data, we concatenate first three parallel states of the input pools and try to compress these three concatenated files. For the compression process, we again use the *best compression* algorithm of WinRAR v4.0.1. First parallel state is compressed with the ratio of 0.07. This can be expected, because input pool is nearly all zero and unchanged. Second and third parallel states are not compressed; i.e. their compression ratio is one. As it can be seen in Figure 4.3, first parallel state is compressed while the second and third parallel states are not compressed. This can be translated as after second mixing, there will be no apparent correlation between parallel runs of the input pool. As a result, it can be said that it is difficult to find correlation and similarities between independent runs.

Parallel States	Total Size (byte)	Compressed Size
Parallel State 0	36864	2591
Parallel State 1	36864	36864
Parallel State 2	36864	36864

Table 4.3: Comparison of normal and compressed (with WinRAR's *best compression*) sizes of the first three concatenated parallel pool states. As it can be seen, after second state, input pools are no longer similar because there is no compression.

4.3 Evaluation of the Results

In this chapter, we compared Linux entropy estimator with two different entropy estimation methods. First method was based on Shannon's entropy function. In order to apply Shannon function, output values of the source and their occurrence probabilities must be known. This was not possible for our tests; therefore we needed to make an estimation for necessary parameters by using sample outcome obtained through our test mechanism. We applied maximum likelihood estimation on the sample, and then applied Shannon's entropy function on the estimation. Calculating the entropy estimation, these results were compared against Linux entropy estimation. Our findings showed that Linux estimator is pessimistic as it should be; i.e. the ratio between our results and estimation of Linux changed between 3–8. As it was mentioned before, being pessimistic on estimating the entropy is a preferred property for random number generators because they should never overestimate the entropy. Providing less than asserted amount of entropy may lead to weaknesses on secure systems.

The second method was based on compression. There is a negative correlation between compressibility and randomness; i.e. the more random the data is, the less compressible it becomes. During this test, we initially compressed the events data—inputs for the input pool— and observed its randomness, in other words, estimated its entropy. The results were much higher than Linux entropy estimation. Later on, we compressed the concatenated input pool states to detect the correlation between different initializations. At the end, we found that there was no correlation between input pool states which means that Linux provides

unique randomness for each run of the system; i.e. during the simulations it is very difficult to reach the same internal states.

As a result, we conclude that Linux random number generator estimates the entropy in a pessimistic way and it provides distinct states on different runs.

Chapter 5

Conclusion and Future Work

"The generation of random numbers is too important to be left to chance."
—Robert R. Coveyou

In this thesis, we analyzed Android random number generator statically and dynamically in order to diagnose its weaknesses. Initially, historical examples of attacks on different random number generators were given in order to emphasize the importance of the topic. Then the works on operating systems' random number generator were examined generally.

After the preparation phase, static analysis of Android random number generator was done. Source code of operating system and the kernel were examined at first. While investigating the general structure of Android `SecureRandom` class, it was found that the seed for that generator actually comes from Linux random number generator. Thus, it was deduced that the security of Android RNG relies on the security of Linux RNG. After this discovery, Linux RNG was analyzed statically. Its general structure and main components were explained in detail. As a conclusion to this chapter, the differences between the version Android emulator used (v2.6.29) and the latest Linux kernel version on April 30th, 2013 (v3.9.0) were listed and explained.

Statical section of the thesis concluded with code analysis and dynamical section began with the modifications on kernel code. Linux RNG file `random.c` in Android kernel was modified in a way so that its internal states and operation can be observed concretely from the kernel logs. The most vulnerable stage of Linux random number generator was the initialization process. With this in mind, the initialization phase of Android was investigated and it was found that there is not any evident flaw or weakness in the design. Even its security is improved with respect to normal Linux by writing device-specific information to entropy pools.

Having surveyed the Android random number generator during the initialization of the device, its entropy estimator and the performance of entropy estimator were studied. Estimation of operating system was compared with maximum likelihood entropy estimation over time differences and it was found that estimation of operating system is pessimistic with respect to our findings. After this test, considering the relationship between entropy and compression, entropy providing inputs were concatenated and compressed. Then compressed size was compared with the entropy estimation. The results revealed that the estimator of operating system is again pessimistic. Estimating the entropy lower guarantees that randomness is at least at the level of estimation. This prevents the crucial problems related with low entropy.

Beside the analysis of Android random number generator running on the emulator, analysis of Android running on hardware can be performed; because, when the main computer is busy, emulator works in a slower way; i.e. it finishes initialization at a longer time. In order to avoid this effect, we always run the emulator under similar workload on the main computer. Running Android OS on Android device will totally eliminate this effect, because no external impact interrupts the running system.

To conclude, Android random number generator is well designed, improved through the contributions from many developers and updated in accordance with the findings of works that are related to random number generator. For today, security related problems may occur only if it is used in a wrong way. But for

the future, it is important to strengthen the Android RNG with the support of hardware RNG. As the time goes on, when the security requirements become more sophisticated and cost of adding hardware RNG diminishes; it will be possible to see many Android devices with hardware RNG.

Bibliography

[1] B. Dole, S. Lodin, and E. H. Spafford, "Misplaced trust: Kerberos 4 session keys," *Proceedings of the Symposium on Network and Distributed System Security*, pp. 60–70, 1997.

[2] G. Taylor and G. Cox, "Behind Intel's new random-number generator." Website, September 2011. http://spectrum.ieee.org/computing/hardware/behind-intels-new-randomnumber-generator/0 last checked: 27.04.2013.

[3] Intel Corporation, "Intel digital random number generator (DRNG) software implementation guide." Website, August 2012. http://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide last checked: 30.04.2013.

[4] T. Reinman, "On Linux random number generator," Master's thesis, The Hebrew University of Jerusalem, Israel, 2005.

[5] B. Arkin, F. Hill, S. Marks, M. Schmid, T. J. Walls, and G. McGraw, "How we learned to cheat at online poker: A study in software security," tech. rep., The Software Security Group at Reliable Software Technologies (RST), September 1999.

[6] B. Barak and S. Halevi, "A model and architecture for pseudo-random generation with applications to /dev/random," in *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, (New York, NY, USA), pp. 203–212, ACM, 2005.

[7] I. Goldberg and D. Wagner, "Randomness in the Netscape browser," *Dr. Dobb's Journal*, pp. 66–70, January 1996.

[8] Z. Gutterman and D. Malkhi, "Hold your sessions: an attack on Java session-id generation," in *Proceedings of the 2005 international conference on Topics in Cryptology*, CT-RSA'05, (Berlin, Heidelberg), pp. 44–57, Springer-Verlag, 2005.

[9] Debian Security Advisory, "Dsa-1571-1 openssl – predictable random number generator." Website, May 2008. http://www.debian.org/security/2008/dsa-1571 last checked: 17.04.2013.

[10] B. Schneier, "Random number bug in Debian Linux." Website, May 2008. http://www.schneier.com/blog/archives/2008/05/random_number_b.html last checked: 17.04.2013.

[11] fail0verflow, "Console hacking 2010 - PS3 epic fail," in *27th Chaos Communication Congress*, 2010.

[12] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, "Ron was wrong, whit is right." Cryptology ePrint Archive, Report 2012/064, 2012.

[13] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of widespreadweak keys in network devices," *Proceedings of the 21st USENIX Security Symposium*, pp. 206–220, 2012.

[14] L. Dorrendorf, "Cryptanalysis of the Windows random number generator," Master's thesis, The Hebrew University of Jerusalem, Israel, 2007.

[15] L. Dorrendorf, Z. Gutterman, and B. Pinkas, "Cryptanalysis of the random number generator of the Windows operating system," *ACM Transactions on Information and System Security*, 2009.

[16] S. Matthews, "Design and analysis of /dev/random, a pseudorandom number generator," 2005.

[17] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the Linux random number generator," *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pp. 371 – 385, 2006.

[18] S. Kiselev, "Security of the Linux random number generator in embedded systems," Master's thesis, University of Haifa, Israel, 2009.

[19] P. Lacharme, A. Rock, V. Strubel, and M. Videau, "The Linux pseudo-random number generator revisited." Cryptology ePrint Archive, Report 2012/251, 2012.

[20] B. Pousse, "Short communication: An interpretation of the Linux entropy estimator." Cryptology ePrint Archive, Report 2012/487, 2012.

[21] T. Vuillemin, F. Goichon, C. Lauradoux, and G. Salagnac, "Entropy transfers in the Linux random number generator," tech. rep., Project-Team PRIVATICS, September 2012.

[22] M. Mackall and T. Ts'o, "random.c—Linux kernel random number generator." Website. http://www.kernel.org last checked: 30.04.2013.

[23] Android 4.2r1, "Package index for Android developers." Website, April 2013. http://developer.android.com/reference/packages.html last checked: 20.04.2013.

[24] Sun-Oracle, "Java^{TM}cryptography architecture api specification and reference." Website, July 2004. http://docs.oracle.com/javase/1.5.0/docs/guide/security/CryptoSpec.html last checked: 20.04.2013.

[25] Oracle and BEA, "Avoiding JVM delays caused by random number generation." Website. http://blog.watchfire.com/files/androiddnsweakprng.pdf last checked: 24.04.2013.

[26] R. Hay and R. Saltzman, "Weak randomness in Android's DNS resolver," tech. rep., IBM Application Security Research Group, July 2012. http://blog.watchfire.com/files/androiddnsweakprng.pdf last checked: 24.04.2013.

[27] M. Matsumoto and Y. Kurita, "Twisted GFSR generators," *ACM Trans. Model. Comput. Simul.*, vol. 2, pp. 179–194, July 1992.

[28] J. Walker, "Conceptual foundations of the ivy bridge random number generator," in *Institute for Security Technology and Society Talks*, Dartmouth College, November 2011.

[29] T. Ts'o, "[patch 02/12] random: make 'add_interrupt_randomness()' do something sane." Website, July 2012. https://lkml.org/lkml/2012/7/6/587 last checked: 27.04.2013.

[30] M. Hamburg, "Understanding Intel's ivy bridge random number generator." Website, December 2012. http://electronicdesign.com/learning-resources/understanding-intels-ivy-bridge-random-number-generator last checked: 27.04.2013.

[31] M. Hamburg, P. Kocher, and M. E. Marson, "Analysis of Intel's ivy bridge digital random number generator," tech. rep., Cryptography Research Inc., March 2012.

[32] G. Kroah-Hartman, "Driving me nuts - things you never should do in the kernel." Website. http://www.linuxjournal.com/article/8110?page=0,2 last checked: 21.04.2013.

[33] Android Open Source Project, "Initializing a build environment." Website. http://source.android.com/source/initializing.html last checked: 21.04.2013.

[34] J. Walker, "Seeding random number generators," in *Workshop on Real-World Cryptography*, Stanford University, 2013.

[35] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, July, October 1948.

[36] M. Shiga and Y. Yokota, "An optimal entropy estimator for discrete random variables," in *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*, vol. 2, pp. 1280–1285 vol. 2, 2005.

[37] T. Schürmann and P. Grassberger, "Entropy estimation of symbol sequences," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 6, no. 3, pp. 414–427, 1996.

Appendix A

Hardware and Software Information

A.1 System Information

Main system and virtual systems used in this work are listed below:

Main Computer : Packard Bell Easy Note TS11HR
Processor : Intel® Core™ i7-2630QM CPU
@ 2.00 GHz 4 Cores 8 Logical Processors
Installed Memory (RAM) : 8 GB
OS : Windows 7 Home Premium Service Pack 1
System Type : 64-bit Operating System
HDD : 750 GB

Virtual Computer : on VirtualBox
Processor : Intel® Core™ i7-2630QM CPU
@ 2.00 GHz 2 Cores 4 Logical Processors
Installed Memory (RAM) : 3 GB
OS : Ubuntu 12.04

System Type : 64-bit Operating System
HDD : 100 GB

Emulated Android : on Android Virtual Device Manager
Processor : ARM (armeabi-v7a)
OS : Android 4.1.2 - API Level 16
SD Card : 8 GB

A.2 Software Information

Software used on main system are listed in Table A.1.

Software Name	**Last Installed Version**	**What It is Used for**
Oracle VM Virtual Box Manager	4.2.12	to setup and run virtual computer
Android SDK Manager	Revision 20.0.3	to emulate Android devices
Notepad++	6.3.2	to edit text, code etc.
WinMerge	2.12.4	to compare text files
WinRAR	4.0.1	to compress files
MATLAB	R2013a	to parse the log files and make calculations on the data
Microsoft Office Professional Plus 2010	14.0.6129	Word, Excel, PowerPoint
Adobe Acrobat 9 Pro Extended	9.5.4	to read *.pdf files
Hex Workshop	6.6	to open and edit hex files
MiKTEX	2.9	to compile LATEX files
TEXnic Center	2.0 Beta 1	to edit LATEX files

Table A.1: Software used throughout this study

Appendix B

Codes

All codes referring from Section 3.1 are presented below:

```
mkdir ~/bin
PATH=~/bin:$PATH
sudo apt-get install curl
curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo
chmod a+x ~/bin/repo
mkdir WORKING_DIRECTORY
cd WORKING_DIRECTORY
sudo apt-get install git-core
repo init -u https://android.googlesource.com/platform/manifest
repo sync -j1

# If download becomes problematic, then try to disable ipv6
gksudo gedit /etc/default/grub
# change GRUB_CMDLINE_LINUX_DEFAULT="quiet splash" line to
# GRUB_CMDLINE_LINUX_DEFAULT="ipv6.disable=1 quiet splash"
sudo update-grub
```

Code B.1: Android source code download

```
git init
git clone https://android.googlesource.com/kernel/goldfish
cd goldfish/
git branch -a
git checkout -t origin/android-goldfish-2.6.29 -b goldfish
```

Code B.2: Android kernel source code download

```
# assuming that kernel source is downloaded to "goldfish" folder under home
gedit ~/goldfish/kernel/printk.c
# change line 747 from "if (printk_time) {" to "//if (printk_time) {"
# change line 763 from "}" to "// }"
```

Code B.3: Enable printing of timestamp

```
# assuming that kernel source is downloaded to "goldfish" folder under home
gedit ~/goldfish/arch/arm/configs/goldfish_armv7_defconfig
# change line 50 from "CONFIG_LOG_BUF_SHIFT=16" to "CONFIG_LOG_BUF_SHIFT=17"
```

Code B.4: Increase kernel `printk` buffer size

```
# assuming that kernel source is downloaded to "goldfish" folder under home
export PATH=~/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin:$PATH
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=arm-eabi-
make goldfish_armv7_defconfig
# X is the number of processors to run during compilation; parallel jobs
make -jX
```

Code B.5: Compile Android kernel

```
:: open command windows as administrator
:: assuming that installation is completed with default settings
cd C:\Users\USERNAME\AppData\Local\Android\android-sdk\tools
:: assuming that kernel image name is not changed
emulator.exe @Deneme -kernel zImage
```

Code B.6: Run Android emulator with desired kernel

```
:: open command windows as administrator
:: assuming that installation is completed with default settings
cd C:\Users\USERNAME\AppData\Local\Android\android-sdk\platform-tools
:: output kernel logs to log.txt continuosly
adb shell cat /proc/kmsg > log.txt
```

Code B.7: Get kernel logs continuously

```
:: open command windows as administrator
:: assuming that installation is completed with default settings
cd C:\Users\USERNAME\AppData\Local\Android\android-sdk\platform-tools
:: open shell to enter bash commands to Android device
adb shell
:: go to devices directory
cd /dev
:: list the files in devices directory
ls
:: there is no "hw_random", go to the attributes directory
cd /sys/class/misc
:: list the files in current directory
ls
:: no related files, no hardware RNG installed
```

Code B.8: Search for hardware RNG in Android