Generating Random Numbers on Android Devices

Jackson Darrow

***Abstract*—Random numbers make up a cornerstone to modern cryptography. The basic construct of random numbers allows us to create proofs that guarantee properties of cryptographic systems with some level of confidence. In this paper I will cover a full gamut of topics related to generating random numbers on the Android platform and will present some of my own findings and data from real tests carried out on multiple Android devices.**

I. INTRODUCTION

Generating random data is an ongoing hard problem that spans multiple fields of computer science but is notably prevalent for its use in cryptography. Modern requirements for random numbers involve being able to generate numbers on the order billions of bytes per minute. The first problem that comes up is generating any data that is random at all in a machine whose sole purpose is to operate deterministically. This is referred to as gathering entropy. Since computers don't have built in coin flippers, they must use outside signals such as IO events or data from sensors. This problem is more or less solved on larger computers with access to disk drives or specialized hardware, but on our smart phones many of these entropy sources do not exist. A phone that has no wifi connectivity, no user input, and no hard disk has almost no way at all to generate entropy, In this project I hope to take a deep dive into the entropy pool of devices under these conditions and analyze their randomness with various statistical tests.

Next, In order to fully understand the situation behind random numbers in Android we need to look at its various pseudo random number generators(PRNGs). PRNGs as a primitive allow us to to leverage the small amount of entropy we do have, by feeding it through deterministic functions that output streams of data that appears random. The task of creating a pseudo random number generator is extremely difficult and to this day it remains a mystery if true pseudo random number generators can even exist. To make up for this uncertainty, we use various statistical tests that gather evidence on a generators ability to produce data that looks random. In the first half of this paper I will discuss Androids various pseudo random number generators and some of their histories.

II. MOTIVATION AND BACKGROUND

Today the Android platform represents roughly 80% of the world's smartphone market share, almost every single person I have ever met owns a cell phone. We use our cellphones to connect to the Internet, communicate and more. Needless to say these devices are tasked with generating random numbers constantly. Because of their ubiquity and the sensitive data they protect, it is imperative that smartphones are able to generate quality random numbers.

III. PSEUDO RANDOM NUMBER GENERATORS

Android relies on two main functions to generate random numbers: java.util.Random and java.security.SecureRandom. Unix like systems have access to files like /dev/urandom and /dev/random but this is not guaranteed by Android. /dev/random is the only random source listed here that does not make use of a PRNG, this will be of interest to us later in the paper.

A. *java.util.Random*

This package is Javas general purpose random number generator that is not meant to be cryptographically secure. This because of the two big reasons I mentioned before. Its seed comes from system time which is completely deterministic, and it is only 48 bits long. This means that the first call extraction of random numbers only has 2**48 possible values which small enough that a personal computer could run through those possibilities in a reasonable amount of time. The second feature that makes this function not suitable for generating secure random number is its choice of PRNG. Random uses a Linear congruential generator which is defined by a simple linear recurrence relationship. Unfortunately LCGs exhibit severe defects, if you plot the output on an n-dimensional space, the points will arrange on hyperplanes. Figure 1 shows LCG drawing points in 3 dimensional space forming 2d planes. True random data would not have these tendencies. The benefits to using LCG as a PRNG is that it's fast and doesn't consume system resources, using system time as a seed also prevents the program from using up the system's entropy when it's not necessary.

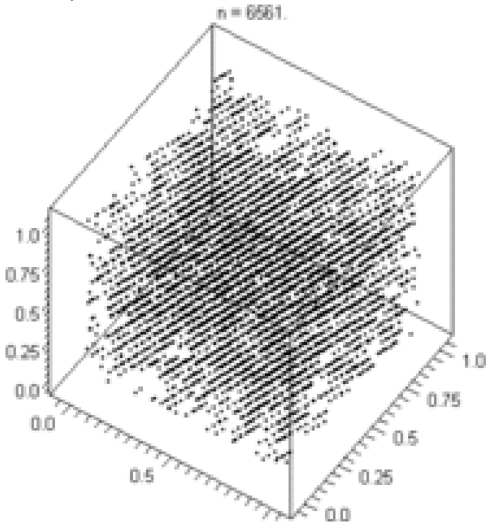

Figure 1

B. *java.security.SecureRandom*

SecureRandom is Androids official cryptographically secure source of random data. SecureRandom has somewhat of an interesting history which I will explain in this section. Up until version 7(Nougat) Android's Java runtime was based on a project called Apache Harmony which became obsolete by SUN's reference implementation in 2006. At this time Android was running their own implementation of Java that used Apache Harmonys version of java.security.*. In android 7(Nougat) Android made an announcement that they were going to be changing SecureRandom[3] due to do various security problems. So what was wrong with SecureRandom? For starters they were using SHA1PRNG as their pseudorandom number generator which was found to produce more 0s than 1s, the problem was even worse for some cases depending on the seed. Despite this flaw SHA1PRNG was still able to make it past the NIST standard for statistical tests for randomness called STS, it also gets the thumbs up from more rigorous tests like dieharder. It took a brand new statistical test to show that SHA1PRNG was broken called LIL which was developed by Yongge Wang[1]. The LIL test works by taking a random walk with the data and checking that the walk behaves on some function bounded with a p-value. A research paper was published in 2013 that exposes weakness in a number of PRNGs including the PRNG involved with infamous Debian purify bug which uses a MD5 construction.

To make this problem even worse, a group of researchers[2] in 2013 discovered a bug in the Android source that caused the SHA1PRNG CTR to overwrite part of the seed leaving only 64 bits of total entropy for the system. This is well within the range of an attacker that is attempting to crack the output of SecureRandom. In the update article released by Android, a developer mentions that many Android apps were using SecureRandom (seeded with the password) as a PBKDF. Out of curiosity I took a look at Google's git repositories for the SHA1PRNG provider, the ones I checked hadnt been updated in 6 years.

As a side note, it is interesting to me that using a hash in CTR mode is a common design for PRNGs, even though hash functions are only required to be collision-resistant and preimage resistant (not random oracles). It sort of seems like we are hoping they also act like PRPs after running a couple of statistical tests on the output. It just goes to show how far we have to come in understanding the underlying mathematics.

IV. Description of Work

For this project I created an Android and Android Wear application that generates random data from various sources on the devices. I tested three devices, a first gen Google Pixel running Android 7.1.1 (Nougat), an LG Power X (6.0.1 Marshmallow) and finally I tested a first gen LG Watch Style running Android 7.1.1 on top of Android Wear 2.0.0. For each of these devices I generated a gigabyte of random data from java.util.Random, java.security.SecureRandom, and /dev/urandom. I then ran the dieharder tests on the data which you can see in figure 3, 4, and 5. I also attempted to run the tests with LIL I could not get to work on my system.

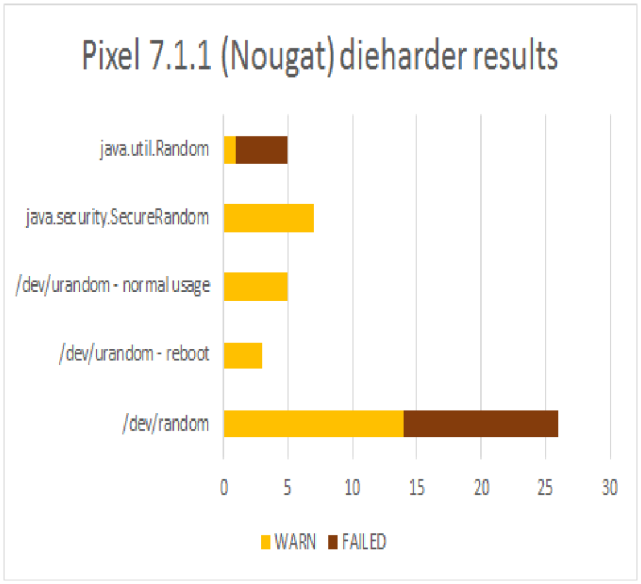

Figure 2

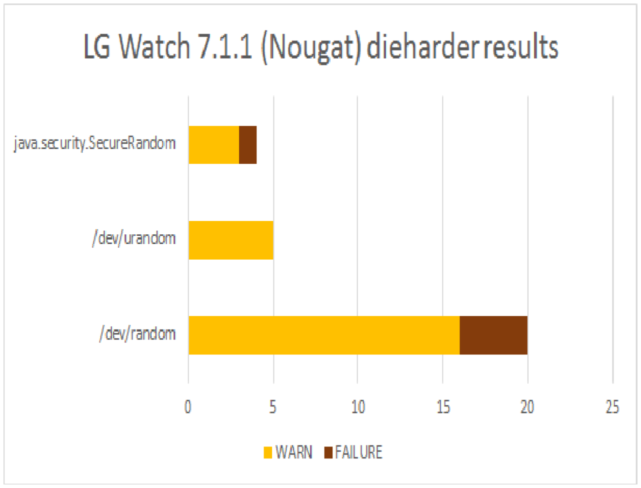

Figure 3

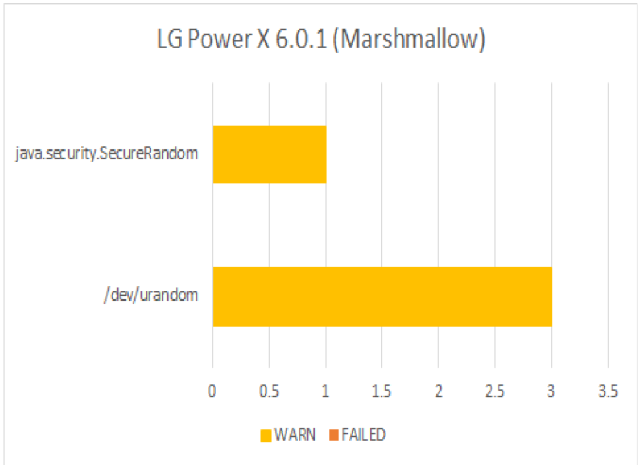

Figure 4

V. Evaluation

From looking at the output from dieharder we see some things that we would reasonably expect and also some things we don't. First of all, the one test I ran on java.util.Random exhibited four FAILING scores from dieharder indicating with high confidence that these numbers are not in fact random. Because of the reason I described earlier, this is an expected result. Moving onto the results from /dev/urandom, we really expect this data to have a negligible difference variance across devices since /dev/urandom is running a pseudo random number generator in the background. It reasonable to think that the reseeding of urandom with higher quality entropy sources in the phones would make the output seem more random. I think that this statement is greatly overestimating the abilities of the dieharder test suite. To support this claim we can take a look at results from SecureRandom. In doing so you might ask yourself "hey isn't Marshmallow's version of SecureRandom provably not random and outputs more 0s than 1s?" I found myself asking the same question, which led to me to wonder how reliable the results from dieharder actually are.

Before we give up all hope lets take a look at the data from /dev/random which is actually measuring the entropy of the android devices as opposed to the quality of various PRNGs. Unfortunately I was not able to easily generate a Gigabite of data from /dev/random from either my watch or cell phone, (the watch kept running out of battery during the test) however I was able to run the dieharder tests with at least 100Mb which is a sufficient since none of any individual dieharder test need more than 65Mb. I was very surprised by the results of these tests since /dev/random is supposed to be the most random source of data. You can see on the graph that the Wear's /dev/random failed 4 tests and weak scores on 16 tests. These results are worse than the output of java.util.Random which as we know is provably not random. After looking into this more, it seems that almost all of the negative results came from a test called rgb_lagged_sum where it failed 2/31 and were weak on 15/31. In the dieharder documentation these tests "test for lagged correlations – the possibility that the random number generator has a bitlevel correlation after some fixed number of intervening bits." [5] This indicates that there was some sort of long period repeating sequence in the random data. If given the time to carry out further testing I would try to find the period and analyze the correlation of the different chunks. It would also be interesting how these correlations change under different testing conditions since the repeating sequences most likely relate to the sources of entropy which is in this case the incoming packets (and possibly others).

Why does /dev/random need to be perfectly random? /dev/urandom is a pseudo random number generator that is constantly being reseeded with data from /dev/random, within a short amount of time the seed space on /dev/urandom becomes so huge it's no exactly critical that /dev/random is 100% random, you can really say that so long as your seed space for /dev/urandom (constantly growing) * the % randomness of /dev/random is above some threshold, you are generating sufficiently good data. Just make sure people never use /dev/random.

VI. Conclusion

When I started this project, I was extremely skeptical that I would find anything shaky about random numbers on Android. Needless to say, we went over many shaky pseudo random number generators and showed that /dev/random is not actually that random. If I were to continue with this project, I would look into how specific apps that use crypto, like Signal, are generating random numbers.

References

[1] Yongge Wang and Tony Nicol *Statistical Properties of Pseudo Random Sequences and Experiments with PHP and Debian OpenSSL* Computers and Security (53):44-64, September 2015

[2] Kai Michaelis, Christopher Meyer, Jrg Schwenk *Randomly Failed! The State of Randomness in Current Java Implementations*

[3] Sergio Giro *Security "Crypto" provider deprecated in Android N* https://android-developers.googleblog.com/2016/06/security-crypto-provider-deprecated-in.html

[4] https://www.random.org/analysis/

[5] http://www.phy.duke.edu/ rgb/General/dieharder.php