Radboud University

Faculty of Science

Additional source of entropy as a service in the Android user-space

Master Thesis

B.C.M. (Bas) Visser

Supervisors:
dr. P. (Peter) Schwabe
dr. V. (Veelasha) Moonsamy

Nijmegen, July 2015

Abstract

Secure encrypted communication on a smartphone or tablet requires a strong random seed as input for the encryption scheme. Android devices are based on the Linux kernel and are therefore provided with the `/dev/random` and `/dev/urandom` pseudo-random-number generators.
`/dev/random` generates randomness from environmental sources but a number of papers have shown vulnerabilities in the entropy pool when there is a lack of user input.
This thesis will research the possibility to provide additional entropy on the Android platform by extracting entropy from sensor data without the need for user input. The entropy level of this generated data is evaluated and this shows that it is feasible to extract randomness from the sensor data within reasonable time and using limited resources from the Android device. Finally, a prototype Android app is presented which uses generated sensor data as input for a stream cipher to create a pseudo-random-number generator in the Android user-space. This app can then be used as an additional source of randomness to strengthen random seeds for any process in the Android user-space on request.

Acknowledgements

First I would like to thank my supervisor, Peter Schwabe. I started with a number of research projects that did not align with my interests and expertise. But when I started this project under Peter's supervision it was a perfect fit for me. It combined my interest in the subject of randomness with my experience in working with the Android platform. I appreciate Peter's patience while I searched for a project fit for me, and during the parts of the research where progress slowed down a bit. Peter has provided me with any resources I needed throughout my research, in the form of knowledge and expertise as well as hardware for building the prototype.

Secondly, I would like to thank Veelasha Moonsamy for being my second supervisor and for taking the time to read and review my thesis.

I would also like to thank my family for supporting me throughout my education and for reviewing the draft version of my thesis. During the more difficult parts I could always come to them for support and advice, which is appreciated.

Contents

Chapter 1

Introduction

In our modern era almost everyone owns a smartphone and for most people this piece of technology has become an essential part of their life. According to an estimation [18] there are currently around 1,9 billion smartphone users around the world, which is over 25% of the world's population. This number is expected to keep increasing rapidly in the coming years. For a device with so many users worldwide it is important that it is properly secured. Many smartphone users will keep personal data on their phones, so a security leak across a large platform like Android would have a heavy impact on society.

However, a smartphone does bring possible security vulnerabilities. A malicious party could intercept or alter smartphone communication. This would have a serious impact on the security and privacy of the smartphone user. For example, a malicious party could listen in on the communication between a smartphone user and the bank. This would cause secure bank information to be compromised. According to an article by The Guardian [1] hackers are actively targeting financial smartphone apps. A significant number of financial apps have been hacked and malicious versions of the app have been uploaded to Google Play or third party app stores. Unsuspecting users who use these malicious apps risk their confidential credentials being captured, or their smartphones being exposed to adware.

1.1 Thesis Goal

In order to prevent these scenarios we need to create a simple way for an Android device to encrypt data and engage in secure encrypted communication. This would be a relatively simple system to build on a larger desktop system because it has practically unlimited resources to work with in terms of computational power and data storage space. However, for a similar system to work on an Android device it will need to take into account the limited amount of resources the device has in terms of battery, computational power and data storage space compared to a desktop system. If the encryption process takes up too much resources, it will interfere with the device user's normal phone activities.

A crucial step for encrypted communication to take place is the generation of a strong random seed as input for the encryption scheme. Ideally the generation of this random seed would take place on the device using a limited amount of computations, battery power and storage space. In this thesis I present a strong lightweight randomness generator prototype in the Android user-space. On boot-up of the Android phone the prototype generates an entropy pool from noise in the phone's sensor data. Once the prototype has generated a sufficiently strong entropy pool, other processes on the Android phone are able to request the prototype for any size of random output. Before presenting the prototype, I analyze the sensor data and estimate the required time to run the data generation process on device boot-up before the entropy pool can be considered

sufficiently strong. For this purpose I have built an app which generates a large amount of data from different device sensors. The goal of the randomness generation prototype is to provide a source of strong randomness which can be used in addition to randomness from existing sources like `/dev/random`. This combination will create a stronger and more secure random seed for encryption processes on the device.

1.1.1 Research questions

The main question of this research is:

How can Android device sensors be used to efficiently obtain high-entropy randomness for cryptographic purposes?

The sub questions for this research are:

1. Which device sensors or combination of device sensors are suitable for extracting high-entropy randomness?
2. How can high-entropy randomness be extracted from the obtained device sensor data and which randomness extraction method is best suited for this purpose?
3. How can the extracted data from the device sensors be evaluated for their entropy quality?
4. Is it feasible to efficiently extract high-entropy random data from device sensors for use in an operational environment?
5. How can the results from this research be used to solve security vulnerabilities which Android device users are confronted with?

The main question will be answered in Chapter 3. Question 1 will be discussed in the related work section, which is Section 1.3. Questions 2 and 3 will be answered in Chapter 3. Questions 4 and 5 will be answered in the conclusion and future work chapter, which is Chapter 4.

1.2 Motivation

Throughout the years there have been many examples of cryptographic systems which are broken by an attack on their random-number generators (RNGs) and random seeds. The consequences of these attacks range anywhere between a minor security leak and a total security disaster. I will use some of these examples to illustrate the importance of having a good source of randomness and therefore the relevance of my research on methods to strengthen randomness on Android.

1.2.1 Debian OpenSSL vulnerability

In 2008, security researcher Bello discovered that the RNG in Debian's OpenSSL package was predictable [5]. This vulnerability was caused by a package update made in 2006 by a Debian developer. The developer removed a section of code used in the seed generation process of the RNG after getting a warning from the analysis tool Valgrind that this code was redundant [33]. The research later revealed that the section of code the developer removed was vital for the security of the cryptographic system because it was responsible for mixing in random data into the seed. This caused the OpenSSL RNG seed to be solely based on the current process ID, which has a maximum of 32.768 by default on the Linux platform. With only 32.768 possible seeds it meant that the seed had become highly predictable and any OpenSSL architecture that used this faulty RNG had become vulnerable for attacks.

Shortly after the discovery in 2008 Debian patched the code and prompted a worldwide regeneration of all keys generated by the faulty RNG [14] [37]. However, to this very day there are still systems in the wild that use the compromised keys. Cox, an engineer at CloudFlare reported that official GitHub repositories of the UK government, Spotify and Python were accessed using compromised SSH keys that were generated using the faulty Debian RNG [13]. Since this report GitHub has started revoking these vulnerable SSH keys from their users and have informed them of the vulnerability.

1.2.2 Playstation 3 compromised signatures

In 2010, a hacker group called fail0verflow (Bushing, Marcan, Segher, Sven) reported that they recovered the ECDSA (elliptic-curve digital-signature algorithm) private key that is used by Sony to sign software for their Playstation 3 game console [11]. This vulnerability was caused when Sony failed to generate a new random nonce for each signature. Because they used the same nonce for multiple signatures they revealed information about their private key and eventually the hacker group recovered the full key, which caused them to be able to sign any software for the Playstation 3. So using this exploit, copied and unlicensed games could be downloaded on any Playstation 3 and the user might not suspect it because it would show as officially signed by Sony. Even worse would be malware showing as officially signed software by Sony because of this exploit, which could infect the systems of unsuspecting Playstation 3 users.

1.2.3 Weak keys in network devices

In 2012, a paper by Heninger et al. showed a widespread flaw in TLS (Transport Layer Security) and SSH (Secure Shell) servers involving weak security keys [22]. They show that malfunctioning RNGs produce low-entropy randomness for the RSA and DSA key generation process, which causes the private keys to be compromised. The researchers attributed this vulnerability to a boot-time entropy hole in the Linux RNGs (`/dev/random` and `/dev/urandom`). During boot-up `/dev/random` uses data left over from the previous boot for the entropy pool, but when the system has been power off long enough for memory to return to its ground state this data is predictable.

The researchers also show that the entropy pool of `/dev/random` lacks entropy when there is a lack of human input events, which confirms the findings in other research such as the papers by Dodis et al. [16] and Voris et al. [40] which I will mention in more detail in Section 1.3.

Lastly, the researchers show that when the Linux kernel extracts entropy from a pool, it hashes the pool contents and mixes part of the result back into the pool. When multiple threads extract entropy concurrently, this creates significant entropy because of the unpredictability in the concurrency behaviour. However, the researchers show that when the kernel is forced to use only one physical thread, this method generates no entropy for the pool.

The vulnerabilities in the randomness generation caused a large number of TLS certificates and SSH keys to be easily factorable and therefore they were compromised. The researchers show that some RSA private keys were obtained because their public keys shared nontrivial common factors due to these entropy problems. Obtaining these security keys compromises the entire TLS and SSH systems, and therefore this is an alarming problem with a large impact on the servers using these compromised systems. Their research shows the importance of having high-entropy randomness during cryptographic key generation, and the importance of solving the problem of low-entropy events in the Linux RNGs.

1.2.4 Taiwanese Citizen Digital Certificate flaw

In 2013, Bernstein, Chang, Cheng, Chou, Heninger, Lange and van Someren presented a paper [9] during Asiacrypt 2013 where they showed that official citizen identification smartcards issued by the Taiwanese governments were flawed. Their research builds on the research by Heninger et al. [22] on low entropy security keys. Where Heninger et al. studied the Internet domain for low entropy security keys, this paper examines if similar flaws can be found in the Taiwanese "Citizen Digital Certificate" database. The researchers investigated 2 million 1024-bit RSA keys from the Taiwanese "Citizen Digital Certificate" database and found that 184 of these keys were trivial to factor in a matter of hours. They attributed these weak RSA keys to a fatal flaw in the hardware RNG. The randomness used for the RSA key generation contained insufficient entropy and created predictable patterns and shared RSA primes.

The researchers showed that the smartcards containing this fatal flaw in their key generation were used in the wild by Taiwanese citizens. The smartcards were used for multiple security sensitive processes such as:

- filing personal income taxes,
- updating car registration,
- making transactions with government agencies (property registries, national labor insurance, public safety and immigration),
- filing grant applications, and
- interacting with companies (e.g., Chunghwa Telecom).

So this flaw could have been a target for malicious parties since there is a significant amount of money involved. Since anyone that can factor the primes of the RSA key has completely compromised the underlying key, they could then forge the smartcard holder's digital signature and steal his identity. Smartcards getting attacked this way on a large scale could have had a severe impact on the Taiwanese society.

1.2.5 Android Bitcoin wallet vulnerability

In 2013, a vulnerability was revealed in the Java library `java.security`, in the class SecureRandom. This flaw caused SecureRandom to have colliding values, and as a result it could generate the same output twice. This meant that SecureRandom had a large entropy flaw because the output had become predictable. Because of this, algorithms that depend on SecureRandom to generate keys or other cryptographic randomness were also compromised.

One of these dependent algorithms was the Android Bitcoin wallet. The Android Bitcoin wallet uses the elliptic-curve digital-signature algorithm (ECDSA) to sign Bitcoin transactions, which is the same algorithm used by Sony to sign the Playstation 3 software as mentioned in Section 1.2.2. The ECDSA signature algorithm was using the SecureRandom Java class on Android devices to generate a random number for each signature. Because of the security flaw, the same random number could be generated for two different signatures. Using the same random value for the ECDSA signature algorithm twice is a total compromise of the security, because an attacker can then easily recover the private key as shown in the example by Nils Schneider [32]. Using the private key, an attacker can sign any transaction and therefore steal Bitcoins from the affected Bitcoin wallet.

This vulnerability had a substantial impact because all Android users with Bitcoins stored on their Android device were at risk of having their Bitcoins stolen by attackers exploiting this security flaw. After this vulnerability was revealed, Google and Bitcoin have released a security update [10]

to fix this problem. After this fix, the Android Bitcoin wallet implementation uses `/dev/urandom` to directly generate randomness for their signatures, instead of using the SecureRandom Java class.

1.3 Related Work

This section will cover relevant research that has already been done in the research area of randomness, entropy and Android device sensor data. This should give an accurate overview of the previous research and where this thesis stands in the research area.

1.3.1 Entropy estimation

Entropy estimation is a vital part in building a pseudo-random-number generator (PRNG) because being able to give an accurate estimation of the amount of entropy contained in the entropy pool is required to reach a certain level of security. If the accuracy of the entropy estimation of a PRNG is high, it can give better security guarantees about the unpredictability of its entropy pool. This makes it less likely for an attacker to compromise the randomness of the PRNG system.

It is difficult however to give a good estimation of entropy, because in a sense this means one has to 'predict' the unpredictability of an entropy pool. A lot of research has been done on the subject of entropy estimation, but the golden solution for fully reliable entropy estimation has not been found yet.

In an article by Watson [42] he proves that the complexity of estimating the min-entropy of a distribution is SBP-complete, which stands for "small bounded-error probability" which is a custom class of complexity that is believed to be equal to NP-complete complexity. So the problem of proving an entropy pool to be truly random is computationally hard, so instead an estimation has to be made using indirect measures.

In a paper by Dodis, Shamir, Stephens-Davidowitz and Wichs [17] they show that many RNGs assume that their internal state is initialized with truly random seeds and that it remains secret at all times. They note that a seed can be compromised after low-entropy events and that most RNG systems remedy this by adding additional randomness from the environment. However, this method of recovering from a state-compromise attack has not been proven to be correct or computationally optimal. In their paper they formalize the problem of designing an efficient recovery mechanism after a state compromise. They assume that any output generated since the state compromise is also compromised until the state becomes truly random again. Their goal is to design a recovery mechanism that stops generating output after state compromise until the state is truly random again. But they also want to make this recovery efficient by making the time between compromise and recovery as low as possible. They deliver a theoretical method for construction of an RNG which meets these goals, which is based on the design of the Fortuna RNG.

The problem stated by the paper is an interesting problem, but it assumes a theoretical scenario that might not be realistic in a practical usage scenario. Especially for the usage scenario of this thesis (i.e., Android devices) the scenario described will be unlikely to cause serious problems. The scenario they describe includes a PRNG that is continuously being used after it has been compromised. On the Android devices, the entropy pool is refreshed every time the phone reboots, which will be quite often for the average user. Additionally, the entropy pool could be set to refresh after certain intervals if the state compromise attacks would pose a serious threat. Implementing the solutions suggested by the paper on the Android devices would reduce the efficiency of the PRNG on a platform which already has to deal with the limited resources provided by the device. Taking these points into consideration, it would not be worth implementing these solutions at this time.

Instead, solutions should be found to provide additional entropy on the Android devices so that the internal state is much less likely to be compromised because of low-entropy events in the first place. The RNG should continuously provide the user with strong random seeds for use in cryptographic protocols on the device, while still remaining as efficient as possible using the limited resources that are offered by the device. The prototype in this thesis achieves this by providing a large amount of additional entropy from the sensor data while still maintaining a high efficiency for the user.

1.3.2 /dev/random

The current standard for randomness generation on Android devices is the `/dev/random` Linux kernel PRNG. `/dev/random` was originally implemented for Linux in 1994 by Ts'o [36]. The `/dev/random` PRNG generates an entropy pool from a number of sources on the hardware level such as inter-keyboard timings and inter-interrupt timings. These sources of entropy are assumed to be non-deterministic and hard for an outside observer to measure. Once sufficient randomness is mixed into the `/dev/random` entropy pool it will accept requests for random bytes and provide these by taking the SHA hash of the contents of the entropy pool.

The Linux kernel also provides a second PRNG which is `/dev/urandom`. `/dev/urandom` is identical to `/dev/random` in its functionality, the only difference is that `/dev/urandom` is non-blocking and has no limit to the amount of requests for bytes of randomness it can take. Since it is considered computationally infeasible to derive any useful information about the input of the SHA hash from its output, `/dev/urandom` will still guarantee cryptographically strong randomness. This is sufficient for many applications, but for applications that require a higher guarantee of randomness it is advised to use `/dev/random` instead. The reason for this is that `/dev/random` will only return a maximum number of bits of randomness based on an entropy estimation of the entropy pool, after this amount has been reached the PRNG will block until it has refilled its entropy pool.

The `/dev/random` PRNG is widely used by most applications and generally considered secure. However, `/dev/random` has been criticized by a number of papers which claim that it has vulnerabilities. Even the source code of `/dev/random` states a weakness with regards to predictability on system start-up. During system start-up the sequence of actions are predictable by an adversary, since there is little to no interaction with a human operator during this time. This can cause the unpredictability of the bits in the entropy pool to drop below the minimal entropy threshold and create a vulnerability. In order to counteract this vulnerability a solution is also provided in the source code. It suggests to carry entropy pools across shut-downs and start-ups using the following scripts:

```
echo "Initializing random number generator..."
random_seed=/var/run/random-seed
# Carry a random seed from start-up to start-up
# Load and then save the whole entropy pool
if [ -f $random_seed ]; then
    cat $random_seed >/dev/urandom
else
    touch $random_seed
fi
chmod 600 $random_seed
dd if=/dev/urandom of=$random_seed count=1 bs=512
```

Listing 1.1: Script for system start-up

```
# Carry a random seed from shut-down to start-up
# Save the whole entropy pool
echo "Saving random seed..."
random_seed=/var/run/random-seed
touch $random_seed
chmod 600 $random_seed
dd if=/dev/urandom of=$random_seed count=1 bs=512
```

Listing 1.2: Script for system shut-down

In 2013 Dodis, Pointcheval, Ruhault, Vergnaud and Wichs also criticized `/dev/random` in their paper [16] claiming that `/dev/random` is not robust. According to them `/dev/random` contains vulnerabilities in the entropy estimator and the internal mixing function. Their proof for the vulnerability in the entropy estimator consists of two ways the estimator can be fooled. First, they show that it is possible to define a distribution of zero entropy that the estimator will estimate of high entropy, as shown in Figure 1.1. And secondly, they show that it is possible to define a distribution of arbitrary high entropy that the estimator will estimate of zero entropy, as shown in Figure 1.2. The reason for this is that the estimator considers timings of the events to estimate their entropy and therefore regular events with unpredictable data are estimated with zero entropy and irregular events with predictable data are estimated with high entropy. In the paper the researchers prove these claims as follows:

Lemma 3. *There exists a stateful distribution $\mathcal{D}_0$ such that $\mathbf{H}_\infty(\mathcal{D}_0) = 0$, whose estimated entropy by* LINUX *is high.*

Proof. Let us define the 32-bits word distribution $\mathcal{D}_0$. On input a state i, $\mathcal{D}_0$ updates its state to $i+1$ and outputs a triple $(i+1, [W_1^i, W_2^i, W_3^i]) \xleftarrow{\$} \mathcal{D}_0(i)$, where $W_1^0 = 2^{12}, W_1^i = ||\lfloor \cos(i).2^{20} \rfloor| + W_1^{i-1}$, $W_2^i = W_3^i = 0$. For each state, $\mathcal{D}_0$ outputs a 12-bytes input containing 0 bit of random data, we have $\mathbf{H}_\infty(\mathcal{D}_0) = 0$ conditioned on the previous and the future outputs (*i.e.* $\mathcal{D}_0$ is legitimate only with $\gamma_i = 0$ for all i). Then $\Delta_i > 2^{12}$ and $H_i = 11$. □

Figure 1.1: from: Dodis et al. [16], proof of their first claim

Lemma 4. *There exists a stateful distribution $\mathcal{D}_1$ such that $\mathbf{H}_\infty(\mathcal{D}_1) = 64$, whose estimated entropy by* LINUX *is null.*

Proof. Let us define the 32-bits word distribution $\mathcal{D}_1$. On input a state i, $\mathcal{D}_1$ updates its state to $i+1$ and outputs a triple: $(i+1, [W_1^i, W_2^i, W_3^i]) \xleftarrow{\$} \mathcal{D}_1(i)$, where $W_i = i, W_2 \xleftarrow{\$} \mathcal{U}_{32}$ and $W_3 \xleftarrow{\$} \mathcal{U}_{32}$. For each state, $\mathcal{D}_1$ outputs a 12-bytes input containing 8 bytes of random data, we have $\mathbf{H}_\infty(\mathcal{D}_1) = 64$ conditioned on the previous and the future outputs (*i.e.* $\mathcal{D}_1$ is legitimate with $\gamma_i = 64$ for all i). Then $\delta_i = 1$, $\delta_i^2 = 0$, $\delta_{i-1}^2 = 0$, $\delta_i^3 = 0$, $\Delta_i = 0$ and $H_i = 0$. □

Figure 1.2: from: Dodis et al. [16], proof of their second claim

In a similar way they show that it is possible to define a distribution of arbitrary high entropy for which the mixing function does not increase the entropy of the internal state. They do note that these attacks against the Linux PRNG show that it does not satisfy the 'robustness' notion of security, but that it still remains unclear if these attacks lead to actual exploitable vulnerabilities in practice.

There have been a number of proposals for alternatives for the Linux PRNG. One of these alternatives is the Entropy Gathering Daemon (EGD) [41], which is a user-space daemon which provides cryptographic random data. Any program that needs random data can connect to an EGD socket

and request random bytes from it. EGD fills its entropy pool by collecting randomness from the output of system statistics programs like 'w', 'last' and 'vmstat'. The main issue with EGD is that it is dependent on operating on a reasonably busy system in order to get strong unpredictable random data from these statistics. This makes it less suitable for use on an Android device, since these devices can be idle for a long period of time which could cause the EGD entropy pool to run out of randomness.

A second alternative is proposed by Intel [23]. Intel approached Ts'o, the developer of the `/dev/random` PRNG, and proposed him to have `/dev/random` rely on the RDRAND instruction. The RDRAND instruction is an Intel on-chip hardware random-number generator which generates random bits from the hardware and feeds this to an AES conditioner which is then used as a seed for the CTR-DRBG DRNG. This DRNG then provides cryptographically secure random numbers to applications requesting them via the RDRAND instruction. Hardware random-number generators generally provide very high entropy because their entropy generation is executed at such a low level that it becomes highly unpredictable. However, the RDRAND Intel hardware RNG is criticized by Ts'o and by Bernstein in his cr.yp.to blog [8] because it uses a proprietary on-chip implementation that cannot be audited. Not only does this mean that the basic functionality of the implementation cannot be checked for flaws or vulnerabilities, but there could also be hidden backdoors in the implementation which can change the entropy strength or even reveal the seeds. These backdoors are not out of the question as shown in 2013 when the New York Times published an article [28] where they state that the National Security Agency (NSA) of the U.S. has been working together with chip makers to insert backdoors in encryption chips. Ts'o has responded to this by saying: *"I am so glad I resisted pressure from Intel engineers to let `/dev/random` rely only on the RDRAND instruction."*

In 2014, Ts'o has updated the `getrandom()` Linux system call to block by default when the entropy pool does not contain at least 128 bits of entropy [35]. Before this update, applications had no way of knowing if the randomness they requested from `/dev/urandom` met a minimal security threshold, because `/dev/urandom` would always be non-blocking and provide randomness even when the entropy pool had not been filled sufficiently yet. After this update, when the entropy pool does not contain at least 128 bits of entropy, any randomness requests made to `/dev/urandom` will block until sufficient entropy has been accumulated. This update has greatly improved the security of the `/dev/urandom` RNG, because it can now provide a security guarantee of providing randomness which contains at least 128 bits of entropy. However, this still means that the Linux RNG is dependent on user input and it might bother the Android device user when it is blocking while it waits until sufficient entropy has been accumulated.

1.3.3 Android device sensor types

There are a number of possible sensor types on Android devices that could be used to extract random bits for the entropy pool. Each of the sensors have different attributes and are different in their reliability when generating random data. Some of these sensors have been researched for their reliability and the entropy level of their generated data.

In 2011 Voris et al. [40] researched the entropy strength of accelerometers and found that they provide a good source of randomness, even when stationary. The researchers have performed a variety of different movements and determined the min-entropy value for each movement sample of 10 minutes, as shown in Figure 1.3 and Figure 1.4. The min-entropy value is calculated using the equation as shown in Section 2.2. The min-entropy values in Figure 1.3 and Figure 1.4 represent the amount of random bits that can be derived from a single 30-bit sensor reading, not the entire distribution sample. The results show that even a stationary accelerometer can provide randomness with a high entropy.

Movement	Min-Entropy
Stationary #1	3.4
Stationary #2	3.6
Hand	10.8
Arc Swipe	11.3
Drop	9.1
Triangle	11.0
Alpha	11.0
Key Twist	11.7
Circle	11.4

Figure 1.3: from: Voris et al. [40], min-entropy estimates of 10 minute motion samples

Movement	Sample Size	Min-Entropy
Overnight #1	1,231,095	3.5
Overnight #2	2,778,113	3.9

Figure 1.4: from: Voris et al. [40], min-entropy estimates of overnight stationary samples

They have also used the NIST Statistical Testing Suite to determine the statistical randomness quality of their results, as shown in Figure 1.5. It shows that all samples almost completely passed the randomness tests as provided by NIST and have a high level of entropy which is cryptographically secure.

Movement	% of NIST Tests Passed
Overnight #1	100.0%
Overnight #2	99.4%
Stationary #1	98.8%
Stationary #2	96.9%
Hand	98.8%
Arc Swipe	98.8%
Drop	97.5%
Triangle	93.8%
Alpha	98.1%
Key Twist	98.1%
Circle	97.5%

Figure 1.5: from: Voris et al. [40], NIST Test Suite results for all samples

Voris et al. also conclude that some sources of entropy used by the Linux kernel like mouse movements and keyboard inputs are dependent on user interaction and therefore not always a reliable source of entropy. They state that entropy generation from the microphone and camera sensors have a similar problem, they can generate data from background noise but they likely need active user interaction to provide reliable entropy. The user might have to create sounds for the microphone or point the camera at different images to provide a higher level of randomness. They also conclude that other automatic sensors like thermometers, proximity sensors and magnetometers are less reliable than the accelerometer but can still provide a strong source of randomness in a similar way.

Chapter 2

Preliminaries

In this chapter I explain the approaches to cryptographic functions such as stream ciphers, hash functions and random-number generators which are essential elements for this thesis. I also describe information about the Android platform and what strengths and weaknesses it has compared to a larger operating system in terms of computational power, memory size, battery and data storage.

2.1 Randomness in Cryptography

In this section I describe approaches to randomness and the role of random data in cryptography. I also describe cryptographic algorithms which are used as building blocks for random-number generator models. Lastly, I describe an attacker model which lists a number of possible attacks on PRNGs and how they are classified based on their properties.

2.1.1 Hash functions

A **hash function** is a mapping h of an input sequence from the alphabet A of arbitrary length to a sequence over A^n of fixed length n, where length n usually ranges between 64 and 512. For cryptographic purposes a length of at least 256 is generally chosen for n to be resistant against brute force attacks, a length of 64 is too small for this purpose. The most important property of hash functions is that it is practically impossible to invert the mapping process.

A **cryptographic hash function** is a hash function that can withstand cryptanalytic attacks. For this purpose it must have the following properties [38]:

- **Pre-image resistance** (i.e., one-way function):

  For almost all outputs b it is computationally infeasible to find an input $a \in A$ such that $b = h(a)$.

- **Second pre-image resistance** (i.e., weak collision resistance):

  For a given value of a it is computationally infeasible to find a second value $a' \in A, a \neq a'$, such that $h(a) = h(a')$.

- **Collision resistance** (i.e., strong collision resistance):

  It is computationally infeasible to find a pair of values $a, a' \in A, a \neq a'$, such that $h(a) = h(a')$.

2.1.2 Stream ciphers

Stream ciphers are encryption algorithms which encrypt a stream of plaintext input using a (pseudo-)random keystream. Each bit of the plaintext input is encrypted individually with a bit from the keystream. There are two types of stream ciphers: in **Synchronous stream ciphers** the keystream is solely based on the random key input, in **asynchronous stream ciphers** the ciphertext output is also fed back into the keystream for the next encryption process [29].

Because of its properties, a stream cipher is well suited for the role of a PRNG in a cryptographically secure pseudo-random-number generator (CSPRNG) construction (see Section 2.1.3 for more information). When used for encryption, a stream cipher combines a keystream and a plaintext to create a ciphertext. When used for randomness generation, the keystream is directly used as a source of randomness. If the stream cipher is fed a high-entropy key from a true-random-number generator, the resulting output stream can be used as a high-entropy CSPRNG. A similar construction will be used in the prototype built for this thesis, extracted sensor data will be combined with a nonce to serve as a key for the Salsa20 stream cipher which is described in a paper by Bernstein [7] after which it can provide random data on request from the output stream.

2.1.3 Random-number generators

Random-number generators are the systems that use a source of entropy to generate random bits of data. Two types of RNGs can be distinguished, **true-random-number generators** and **pseudo-random-number generators**.

True-random-number generators use a source of natural entropy to generate randomness (e.g., background noise, atmospheric noise, electromagnetic hardware noise or cosmic radiation). Natural entropy is generally highly unpredictable, non-deterministic, and hard for an outside observer to measure. The downside of natural entropy is that the RNG needs to refresh its entropy pool regularly to remain unpredictable. During the process of harvesting additional entropy, the RNG is blocking until it has gathered enough entropy to meet demand. `/dev/random` is an example of this process, once it has reached its entropy limit it will block until it has gathered sufficient entropy to refill the entropy pool. This blocking behaviour can severely slow down systems when very large randomness requests are made.

Pseudo-random-number generators uses one-way functions to generate entropy. These algorithms need an input seed in order to generate long random sequences of data. There are many examples of PRNGs like Fortuna [19], the Yarrow algorithm [24] and constructions which use stream ciphers like AES-CTR, XSalsa20 [7] or ChaCha20 [6]. The advantage of PRNGs is that they can generate a practically infinite stream of random data once seeded properly, and will not face the problem of having to block the process to refresh entropy. The downside is that the seed input of the PRNG is a major vulnerability. If the seed leaks, all generated data is compromised and becomes predictable. Therefore it is important that the seed input is not predictable by an outside observer and has a high level of entropy.

Cryptographically secure pseudo-random-number generators are PRNGs adapted for use in cryptography. CSPRNGs need a high-entropy seed to generate randomness of a high-entropy quality, which makes it suitable for use in cryptographic algorithms. One way to achieve this is by combining a true-random-number generator and a pseudo-random-number generator to create a CSPRNG model. In this model a true-random-number generator is used to generate a high-entropy seed from a non-deterministic source (e.g., a hardware random-number generator). This seed is then used as input for a PRNG construction to generate large amounts of random data. If constructed properly, this model has the benefit of being non-deterministic like a true-random-number generator, but it is still able to generate large amounts of data without blocking once the PRNG seed has been initiated. This construction will also be used in the prototype of this

thesis. Android device sensor data will be extracted and hashed to create a high-entropy seed, and this seed will be used as input for the XSalsa20 PRNG construction to be able to generate large amounts of random data on request. When a CSPRNG is constructed correctly (i.e., the entropy source is non-deterministic and the one-way function is secure and securely implemented), it will be suitable for use in cryptographic protocols like key generation for encryption schemes or nonces in authentication protocols.

2.1.4 Attacker model

There are a number of possible attacks on PRNGs, and it is important to build an attacker model when designing a PRNG. Kelsey, Schneier, Wagner and Hall [25] describe three different classes of attack against PRNGs in their paper:

1. Direct Cryptanalytic Attack
2. Input-Based Attacks
3. State Compromise Extension Attacks
   (a) Backtracking Attacks
   (b) Permanent Compromise Attacks
   (c) Iterative Guessing Attacks
   (d) Meet-in-the-Middle Attacks

In a **direct cryptanalytic attack** the attacker is directly able to distinguish between PRNG outputs and random outputs. So an attacker directly exploits a vulnerability in the cryptographic properties of the PRNG system.

In **input-based attacks** the attacker has access to the input of the PRNG and uses knowledge or control over the input to cryptanalyze the PRNG. The attacker can use chosen, known or replayed inputs to manipulate the system.

In **state compromise extension attacks** the internal state of the PRNG is used by the attacker to recover previous outputs. These attacks can be divided in four categories:

- In **backtracking attacks** the attacker uses a compromised state S of the PRNG at time t to reveal previous outputs.
- A **permanent compromise attack** occurs if all future and past values of the internal state S are vulnerable to attack once an attacker compromises S at time t.
- In **iterative guessing attacks** the attacker has knowledge of internal state S at time t and uses this knowledge to learn S at time $t + \epsilon$.
- A **meet-in-the-middle attack** is a combination of an iterative guessing attack and a backtracking attack. The attacker uses knowledge of S at times t and $t + 2\epsilon$ to recover S at time $t + \epsilon$.

2.2 Entropy

In this section the concept of entropy will be explained and it will introduce approaches to entropy generation and entropy estimation for RNG systems.

2.2.1 Entropy and unpredictability

The encyclopedia of mathematics [15] defines entropy as a measure of the degree of indeterminacy of a random variable. If ξ is a discrete random variable defined on a probability space $(\Omega, \mathcal{F}, P)$ and assuming values $x_1, x_2, ...$, with probability distribution $\{p_k : 1, 2, ...\}, p_k = P\{\xi = x_k\}$, then the entropy is defined by the formula:

$$H(\xi) = -\sum_{k=1}^{\infty} p_k \log p_k \tag{2.1}$$

This measure is also known as Shannon entropy, which provides the average case entropy measure for an independent distribution of random variables.

As an example of Shannon entropy, consider a coin flip with two independent fair coins (X,Y) which have exactly 50% chance to flip heads (H) or tails (T). There are then four possible outcome combinations: (T,T), (T,H), (H,T), (H,H) each with a 1/4 chance of occurring. Then the resulting Shannon entropy is $H(X,Y) = 4(-\frac{1}{4}\log_2 \frac{1}{4}) = 2$, so these two coins contain 2 bits of unpredictability.

Now assume the second coin Y is not fair, and instead has a 3/4 chance to flip heads. Then the four outcome combinations have a different chance of occurring: P(T,T)=1/8, P(T,H)=3/8, P(H,T)=1/8, P(H,H)=3/8. Now the resulting Shannon entropy is $H(X,Y) = 2(-\frac{1}{8}\log_2 \frac{1}{8}) + 2(-\frac{3}{8}\log_2 \frac{3}{8}) = 1,8$. So now the two coins only contain 1,8 bits of unpredictability and have therefore become more predictable.

According to a paper by Barak and Halevi [2] and a recommendation by NIST [4] a better entropy measure to use is the min-entropy, also known as Rényi entropy. The min-entropy is defined as follows:

$$H_\infty(X) = \min_{x \in X} (-\log P_X(x)) = -\log (\max_{x \in X} P_X(x)) \tag{2.2}$$

The min-entropy measure of entropy is recommended over Shannon entropy because min-entropy considers the worst case instead of the average case, and is therefore a more restricted case and a better metric for entropy measurement in cryptography.

Another method to measure entropy is the Kolmogorov complexity [26]. This measure tests if a given sequence can be specified using less computational resources than the string itself. If a sequence has repeating patterns, it can be specified in fewer characters than the string itself. In order for the sequence to be defined as random it should not be possible to specify the sequence with fewer characters than the sequence itself.

Entropy can also be measured by testing if a sequence passes Golomb's randomness postulates. These postulates are stated in [38] as follows:

1. The number of zeros and the number of ones are as equal as possible per period, i.e., both are $p/2$ if p is even and they are $(p \pm 1)/2$ if p is odd.

2. Half of the runs in a cycle have length 1, one quarter of the runs have length 2, one eight of the runs have length 3, and so forth. Moreover half of the runs of a certain length are gaps, the other half are blocks.

3. The out-of-phase autocorrelation AC(k) has the same value for all values of k.

Where the *autocorrelation* AC(k) of a periodic sequence $\{s_i\}_{i\geq 0}$ with period p is defined by:

$$\mathrm{AC}(k) = \frac{\mathrm{A}(k)-\mathrm{D}(k)}{p} \tag{2.3}$$

Where A(k) and D(k) denote the number of agreements and disagreements respectively, over a full period between $\{s_i\}_{i\geq 0}$ and $\{s_{i+k}\}_{i\geq 0}$, which is $\{s_i\}_{i\geq 0}$ shifted over k positions to the left.

If a periodic sequence satisfies all three postulates, it can be called pseudo-random. Take for example the sequence `101010101010101010`. This sequence satisfies the first postulate because #1's = #2's = 9. It does not satisfy the second postulate because all runs are of size 1. The autocorrelation of this sequence is as follows:

```
t=0
101010101010101010
101010101010101010 AC = 18 - 0 = 18

t=1
101010101010101010
010101010101010101 AC = 0 - 18 = -18

t=2 (= t=0)
101010101010101010
101010101010101010 AC = 18 - 0 = 18
```

The autocorrelation is not equal on every shift of the sequence, so the sequence also does not satisfy the third postulate. So following Golomb's postulates it can be concluded that the sequence `101010101010101010` is not sufficiently random.

2.2.2 Entropy generation

There are a number of ways to generate entropy from different sources. But first a process is required that is unpredictable and therefore contains a high amount of entropy. Examples of such processes are clock jitter, electromagnetic fields and cosmic radiation. Other examples are keyboard timings and mouse movements which `/dev/random` uses as a source of entropy, system statistics which the entropy gathering daemon uses and sensor-data noise which the prototype for this thesis uses.

These processes contain entropy to provide computational randomness, but might still contain large sections of bad statistical randomness. Therefore the data gathered from these processes will need to be extracted first to provide statistical randomness as well as computational randomness. Data is statistically random if it passes statistical randomness tests such as the NIST statistical test suite [31] and Diehard randomness tests [27]. Computational randomness considers the amount of entropy in data, data is computationally random if it contains sufficient entropy and is therefore not predictable for an attacker. The most used method of randomness extraction is a cryptographic hash function like SHA-2 (Secure Hash Algorithm). This is also the method used in both the `/dev/random` system and the prototype for this thesis. Alternative methods of extraction are the Von Neumann extractor [39], or custom cryptographic extraction functions as described in an article by Barak, Impagliazzo and Wigderson [3].

If the randomness generator needs to provide a constant stream of random sequences it also needs a PRNG to transform a strong random input seed into a constant keystream which provides a practically infinite amount of random output. `/dev/random` only provides randomness directly extracted from its entropy pool and will be a blocking process as it refills its pool when too much

randomness is requested. However, the `/dev/urandom` process uses a stream cipher to provide a non-blocking method of randomness generation, it uses the extracted data from the entropy pool as key input. The prototype for this thesis works in a similar way, using the SHA-256 hash to extract a 256-bit key from the sensor data which then serves as random key input for the XSalsa20 stream cipher algorithm to provide a constant stream of randomness.

2.2.3 Entropy estimation

During the process of entropy generation it is important that the level of entropy in the entropy pool is estimated to provide a certain level of security. It is generally hard to estimate the level of entropy in a real time situation because being able to proof the unpredictability of an entropy pool would make it predictable by definition. However, estimations can still be made based on the indirect properties of an algorithm and the entropy source.

In his article [30] Pousse describes the entropy estimation method of `/dev/random`. `/dev/random` uses event-timing estimations based upon the Kolmogorov complexity as can be seen in the source code in Listing 2.1. If an event (e.g., a mouse movement) happens too soon after the previous event it is estimated as low entropy. Also, it checks if the time between consecutive events is not too predictable. Based on these parameters `/dev/random` will make an on-the-fly estimation of the number of bits of entropy that are in its entropy pool.

```
delta = sample.jiffies - state->last_time;
state->last_time = sample.jiffies;

delta2 = delta - state->last_delta;
state->last_delta = delta;

delta3 = delta2 - state->last_delta2;
state->last_delta2 = delta2;

if (delta < 0)
    delta = -delta;
if (delta2 < 0)
    delta2 = -delta2;
if (delta3 < 0)
    delta3 = -delta3;
if (delta > delta2)
    delta = delta2;
if (delta > delta3)
    delta = delta3;

/*
* delta is now minimum absolute delta.
* Round down by 1 bit on general principles,
* and limit entropy entimate to 12 bits.
*/
credit_entropy_bits(r, min_t(int, fls(delta>>1), 11));
```

Listing 2.1: Entropy estimation of `/dev/random`

In order to make on-the-fly estimations of entropy, these estimations have to be made based on indirect properties like event timings. An alternative is to make a statistical estimation of the entropy by collecting data in advance, this data can then be analyzed using statistical estimations like the Shannon entropy or min-entropy measures. This method of entropy estimation is also used in the prototype for this thesis, a large amount of sensor data has been generated and analyzed in order to make an estimation of the entropy gathered within a given time frame. This estimation is then used to achieve the 256-bit security threshold that is required for the input to the hash function, so that the entropy from the sensor data does not become a bottleneck for the security level of the system as a whole.

2.3 Android Platform

In this section the Android platform will be described and background information about the system with its strengths and weaknesses.

2.3.1 Android operating system

The Android operating system is based on a Linux kernel and developed by Google. Android is designed for touchscreen devices like smart phones and tablets, but it is also used in the new generation of devices like smart televisions, smart cars and smart watches. The user interacts with the Android device by performing motions on the touchscreen like swiping, tapping and pinching. The front-end of the Android operating system consists of Android applications called apps. Each app represents an application which has its own functionality and design. The user can install new apps on the device to expand functionality. Most apps are installed through the Google Play store, which is a cloud service by Google which hosts over a million apps. App developers can publish new apps on the Google Play store which the users can then directly download and install on their devices.

The main issue of Android is that different generations of devices often use different versions of the Android platform. From a developers perspective there are quite a lot of differences between the Android platform versions. Since it is generally hard for developers to support all versions, apps are usually tagged with a minimum supported version. Figure 2.1 shows that over 95% of all Android devices use version 4.0.3 (Ice Cream Sandwich) or higher. Because of this, app developers often choose to support a minimum version of 4.0.3, to cover a large amount of Android users while still excluding outdated versions.

Version	Codename	API	Distribution
2.2	Froyo	8	0.3%
2.3.3 - 2.3.7	Gingerbread	10	4.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	4.1%
4.1.x	Jelly Bean	16	13.0%
4.2.x		17	15.9%
4.3		18	4.7%
4.4	KitKat	19	39.3%
5.0	Lollipop	21	15.5%
5.1		22	2.6%

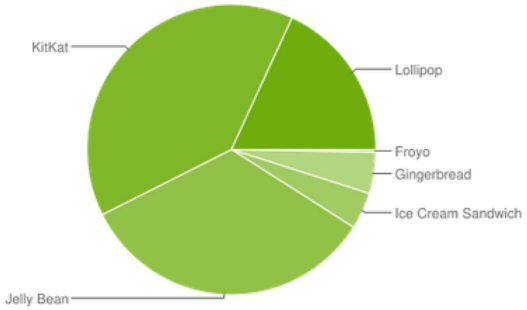

Figure 2.1: from: Android Developer: Dashboards [20], Android platform statistics August 3, 2015.

Developing on mobile devices is different from developing on larger systems, it has a number of strengths and weaknesses. A mobile device developer needs to consider the limited resources that are available on the device. A mobile device is often limited in terms of computational power, memory size, battery and data storage. Since cryptographic algorithms and randomness genera-

tion can use a high amount of system resources, it is important that resource limitations are taken into consideration when developing such algorithms for mobile devices. Randomness generation algorithms might have to use lower-entropy data to limit resource usage, and cryptographic algorithms might have to accept smaller security parameters. Finding the right balance between using limited resources while still maintaining a high level of security on these devices proves a challenge.

The strength of developing on the Android platform is that it is well supported and documented by Google. In 2014, Google released Android Studio, which is an integrated development environment (IDE) specialized in Android development and therefore fully tailored for this purpose. Development is done in the Java programming language, which makes a lot of existing Java frameworks almost fully compatible with the Android system. Another advantage is that development on Android can directly be tested by connecting an Android device to the development environment, this will directly install the app on the device and initiate it for testing.

2.3.2 Kernel and user-space

The Android operating system can run processes in the kernel space and in the user space. Running processes in the kernel space requires the device to be rooted to gain access to the system core. Processes in the user space are more restricted in resources and functionality, while processes in the kernel space have full access to all resources and functionality. Development in the kernel space can introduce security risks if processes are compromised, but these processes do have direct access to hardware which is sometimes required for randomness generation.

There are a number of differences and advantages between running a CSPRNG system in the kernel space or in the user-space. `/dev/random` is an example of a RNG that runs in the kernel space. The advantage of running in the kernel space is that the application has direct access to the hardware and can extract raw values from hardware sensors without interference from higher level layers. Another advantage is that a kernel application does not get interrupted by the process scheduler and can keep running in the background. The disadvantage is that a kernel application could use up a lot of resources and block or slow down the user when high resource actions are performed (e.g., playing videos or music). Unless the user has rooted the device, he can not prevent a kernel application from slowing down the system in this case.

The advantage of running an RNG in the user-space is that it has more direct interaction with the user. The user is aware the process is running and the process is throttled by the scheduler so it does not bother the user when it is running in the background. The disadvantage is that hardware entropy might not be received as raw data at the user-space level since it might have been filtered by lower-level applications. For example, the camera might provide raw camera data on the kernel level, but be subject to pre-processing by a lower-level camera driver when read out on the user-space level. Another disadvantage is that the scheduler might shut down the application unexpectedly when it is using up too much resources, causing possible undefined behaviour and possibly even leaking security sensitive information when not shut down correctly.

The prototype built for this thesis runs at the user-space level to provide the best possible interaction with the user for demonstration purposes. The sensor data used for the entropy generation is obtained through a direct interface which gathers raw high-entropy sensor output. The application is also flagged to run as a foreground process, this gives the application a top priority with the scheduler. This causes the application to practically never be shut down unexpectedly by the scheduler in the general usage scenario. In case the application does get shut down by the scheduler, security measures have been put in place to clear sensitive information before shutting down.

For an actual version of the sensor entropy daemon in the field it would be best to move it to the kernel level. This way it can directly gather entropy from the hardware and feed it to the `/dev/random` process to provide the additional entropy. In this setup the two processes would work together for the maximum amount of security, and the processes would be protected from the scheduler within the kernel level.

Chapter 3

Entropy as a service

3.1 Approach and Implementation

In this section I give a detailed description of the research approach and the design choices made for the implementation of the prototype.

3.1.1 Research approach

The approach for this research consists of two parts. First, an analysis has been made of large samples of Android device sensor data which indicate that the sensor data could contain entropy and can be used for randomness generation purposes. To achieve this, an application has been built that can gather and store large amounts of sensor data under different circumstances. The results of this analysis give an estimation of the amount of entropy which can be extracted from sensor data.

Secondly, to answer the research question of this thesis, I have built a prototype which will serve as a proof of concept. This prototype shows that is it possible to use Android device sensor data to efficiently obtain high-entropy randomness. If the prototype is adapted for use in conjunction with `/dev/random`, then the resulting random data could be used in an operational environment for cryptographic purposes. This conjunction of the existing standard for randomness generation, `/dev/random`, and the proposed generation method using sensor data will strengthen the randomness quality and make it more resistant to attacks.

It is important to note that the proposed method of randomness generation using sensor data and the prototype are not meant to replace `/dev/random`. The prototype is meant to provide additional randomness from other sources to the existing entropy pool of `/dev/random`. The criticism from a number of papers on `/dev/random` is that it is too dependent on user input, and the entropy might be of poor quality and even predictable if the user provides very little or no input during boot-up when the entropy pool is generated. Entropy from sensor data is much less dependent on user interaction and this will help strengthen `/dev/random` on the aspects that have received criticism. This conjunction between the prototype and `/dev/random` will combine the solid foundation of the `/dev/random` hardware entropy extraction with the strong entropy found in sensor data.

3.1.2 Prototype design

The general structure of the prototype is shown in Figure 3.1. The prototype automatically detects which sensors are available for data generation on the device it is used on. The available sensors will periodically generate data and this data is fed to the SHA-256 hash. When sufficient entropy has been gathered from the sensor data, the hash will digest all data into a 256-bit output. This

256-bit output is used as a key for the XSalsa20 algorithm [7], along with a 192-bit incremental nonce. The resulting keystream from the XSalsa20 algorithm is then used as a continuous stream of randomness which can provide randomness on request.

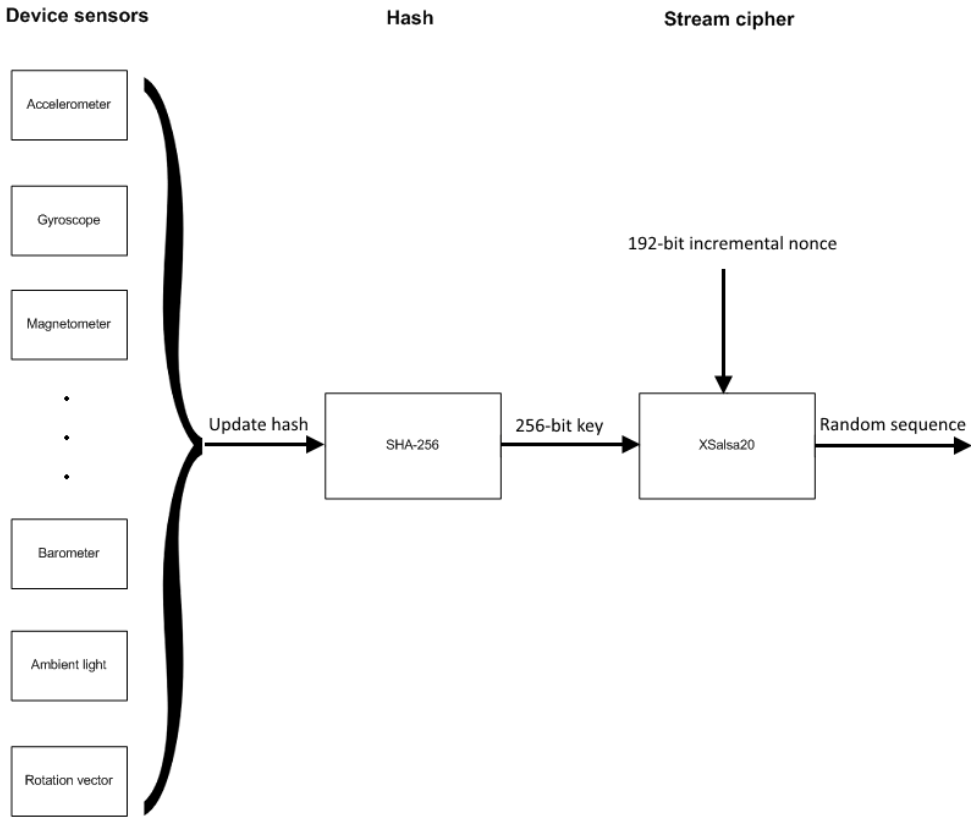

Figure 3.1: Overview of the prototype implementation design

Aside from the back-end construction of the prototype, it also contains a simple front-end user interface for demonstration purposes. The user can interact with the prototype by entering a number of bits of randomness to request. The user will receive information about the time remaining to fill the entropy pool if the sensor data is still being processed. After the system is ready to receive randomness requests, the resulting randomness is displayed to the user in a readable hexadecimal format. Detailed information and technical documentation on the prototype and the user interface can be found in Appendix A.

A number of frameworks are used in the prototype structure:

The Android SensorEvent API is used to listen to sensor events and generate the sensor data. The SensorEvent API can list all sensors available on a device, which are then used for data generation. The API stores all raw sensor data it receives in the *values* array, where each value in the array represents an axis of a sensor. For example, the accelerometer sensor provides the acceleration on the x-axis, the y-axis and the z-axis as shown in Figure 3.2. The rate at which the API generates the sensor data is set to the shortest delay in the prototype, which will generate the highest amount of data possible during the entropy pool generation. The exact amount of data generated varies depending on the device that runs the prototype, but testing shows that there will be around 375.000 sensor readings per minute on average.

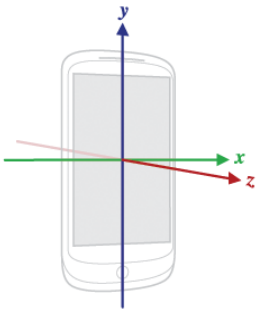

Figure 3.2: from: Android Developers: SensorEvent API [21]

The Java MessageDigest library is used for the SHA-256 hash, this library provides methods to update the hash with data and digest the output once the entropy pool has been filled. This hash provides a 256-bit output and follows the hash properties of one-way function and collision resistance as mentioned in Section 2.1.1.

For the XSalsa20 [7] algorithm implementation the Spongy Castle API [34] is used, this API is a variation of the Bouncy Castle Java Cryptography API [12] which is adapted to work on Android devices. The XSalsa20 algorithm is a modification of the Salsa20 algorithm with a larger nonce size to prevent nonce overflow. The normal Salsa20 algorithm has a 64-bit nonce, whereas the XSalsa20 algorithm has a 192-bit nonce. In the normal use case this difference should not matter since there should never be more than 2^{64} requests per session, but a larger nonce size makes the algorithm more resistant to overflow attacks. With a smaller nonce size, an attacker could make a large number of requests to overflow the nonce, this could cause the counter to reset and cause the same nonce to be used twice. This would break the security because this will reveal the input key and compromise the entire system. The XSalsa20 algorithm automatically increments the nonce after every request it receives, so it will never use the same nonce twice.

3.1.3 Entropy estimation

An important part of building the RNG prototype is that the entropy pool contains sufficient entropy before it extracts the randomness through the hash. Since the SHA-256 hash outputs a 256-bit key, the entropy pool should contain at least 256 bits of entropy. If the entropy pool contains at least 256 bits of entropy, the entropy pool would be at least as hard to brute force by an attacker as the 256-bit key hash output, which makes sure the entropy pool does not become a bottleneck for the RNG security. So to make sure the entropy pool contains at least 256-bits of entropy, an estimation of the entropy produced by the sensors has to be made. However, the paradox of entropy estimation is that it needs an indication of how unpredictable a sequence is. However, if an absolute proof of unpredictability could be made, then by definition the sequence would be predictable.

For the purpose of the prototype I used large samples and based on these samples I give an indication of how unpredictable the output will be in the general case. Figure 3.3 lists a small sample of 17 sensor outputs from the accelerometer of two Android devices, the Nexus 7 and the Nexus 4 with them laying motionless on a desk. At first glance, it looks like the first two numbers and the first decimal are highly predictable, but the second through sixth decimals seem unpredictable. As Figure 3.4 shows, these five decimals signify 17 bits when converted using the IEEE 754 conversion. So in the best case where it is assumed that these five decimals are indeed unpredictable, one sensor output would produce 17 bits of entropy.

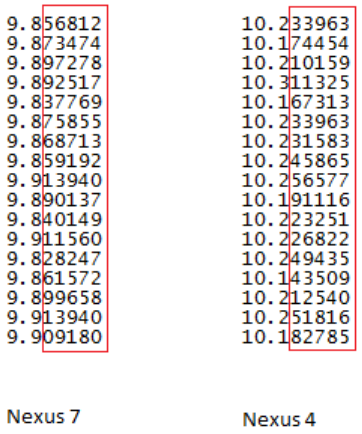

Figure 3.3: Sample sensor data from the Nexus 7 (left) and the Nexus 4 (right)

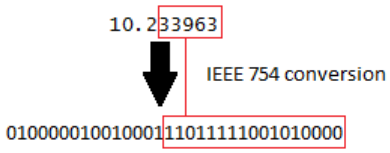

Figure 3.4: Entropy in bits after IEEE 754 conversion, 5 decimals signify 17 bits

Further sampling shows that these five decimals are not equally unpredictable. Table 3.1 shows a large sample of 11787 sensor outputs with the distribution of decimal values of each of the five decimals, this sample was gathered on the Nexus 7 device with it laying motionless on a desk. It can be seen that the third and fourth decimals contain the highest unpredictability, with an almost perfect distribution of all 10 decimal values. The second and sixth decimals are somewhat unpredictable as well, but have a worse distribution than the third and fourth decimals. The fifth decimal is quite predictable, the distribution is irregular with the highest occurring value occurring over three times as often as the lowest occurring value.

The decimal value distributions are shown in more detail in the histograms in Figures 3.5, 3.6, 3.7, 3.8 and 3.9. The histogram of the fifth decimal shows that the distribution runs very irregular from 4.8% up to 14.56%. It seems there is a correlation between odd and even numbers and their occurrence in the distribution, the odd numbers occur about twice as often as the even numbers. This has quite a large impact on the entropy of the decimal, since an attacker will have twice the chance of guessing the decimal correctly if he tries odd numbers. So the entropy of the fifth decimal is only around half the amount of the more evenly distributed decimals.

	Second decimal	Third decimal	Fourth decimal	Fifth decimal	Sixth decimal
0	864	1159	1222	566	1270
1	725	1061	1268	1643	889
2	807	1459	1203	827	1274
3	979	1356	1124	1541	1291
4	1536	1121	1135	841	1316
5	1412	1138	1072	1558	934
6	1422	1071	1110	791	1325
7	1456	1164	1251	1589	1293
8	1286	1113	1200	715	880
9	1300	1145	1202	1716	1315

Table 3.1: Value occurrences of the last five decimals of the sensor data

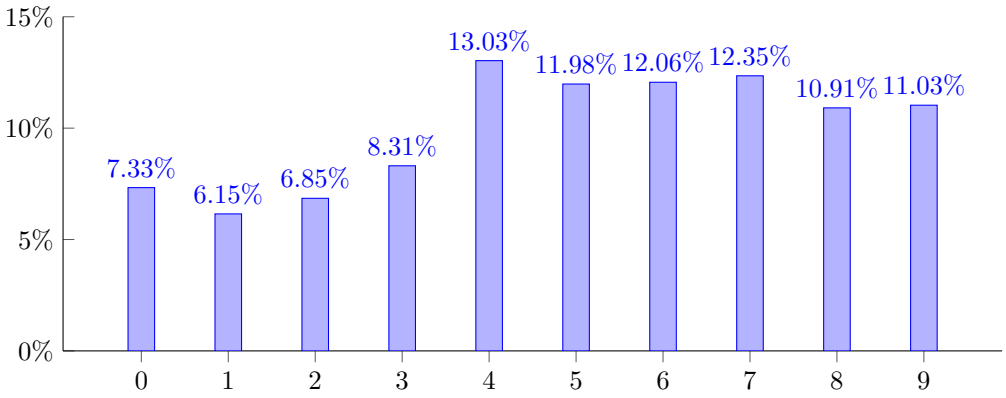

Figure 3.5: Histogram of the second decimal value occurrences

In order to translate these results to an entropy-gathering duration setting for the prototype I distinguish between three scenarios: the worst-case scenario, the average-case scenario and the best-case scenario.

For the worst-case scenario the assumption is made that all decimals except the fourth one are not distributed evenly enough and are therefore completely predictable, and therefore only the fourth decimal provides some entropy. Furthermore, the assumption is made that this single decimal will only affect the least-significant bit after IEEE 754 conversion. Lastly, it is also assumed that the Android device will be laying motionless, just like it was during the generation of the samples. Under these assumptions, one entire sensor reading would only provide 1 bit of entropy for the entropy pool. So in order to reach the 256-bit entropy threshold required for the RNG security, the hash would need to be fed 256 sensor readings. Table 3.2 shows the total amount of sensor readings across different devices and under different circumstances. From this data it can be derived that there are an average of 374600 sensor readings per minute when averaging all 8 session results. So it will take 42 milliseconds to generate 256 sensor readings, which contain 256 bits of entropy.

For the average-case scenario the assumption is made that only the fifth decimal is not evenly distributed enough and therefore contains 0 bits of entropy, so the remaining four decimals all contain some entropy. Secondly, the assumption is made that each of these four decimals will only affect the least significant bit after their IEEE 754 conversion. And lastly, it is assumed that the Android device will be laying motionless, just like it was during the generation of the samples. Under these assumptions, one sensor reading would provide 4 bits of entropy for the

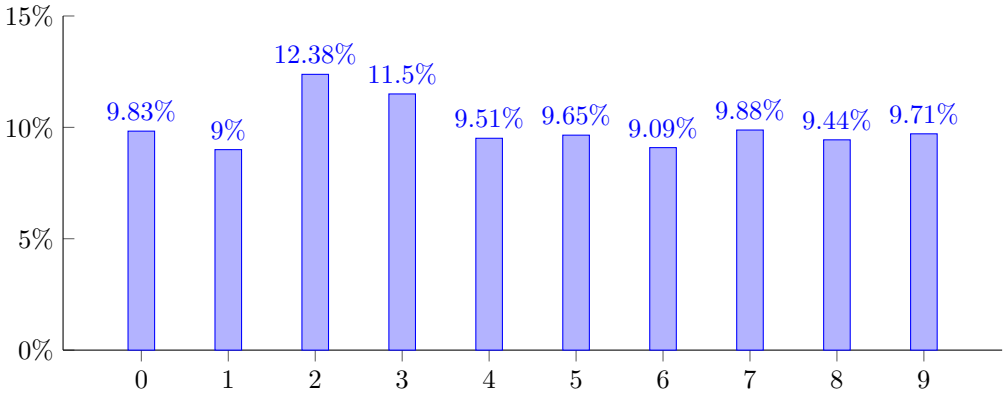

Figure 3.6: Histogram of the third decimal value occurrences

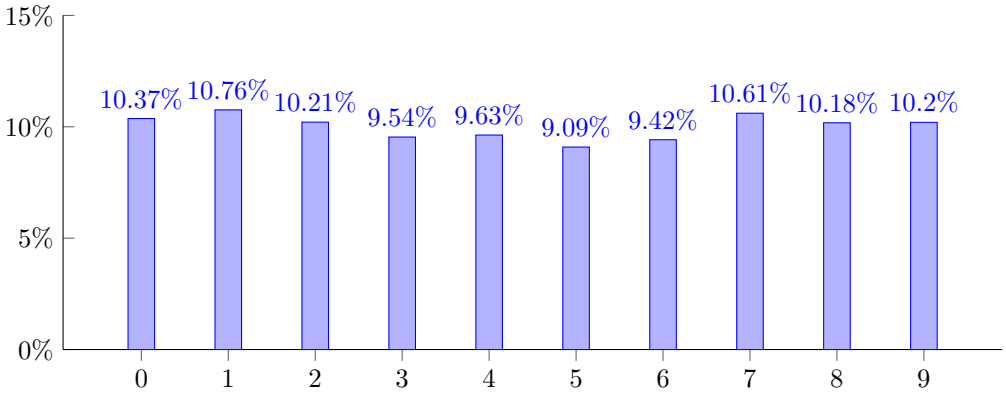

Figure 3.7: Histogram of the fourth decimal value occurrences

entropy pool. So in order to reach the 256-bit entropy threshold, the hash would need to be fed 64 sensor readings. Taking the average of 374600 sensor readings from table 3.2, this results in a duration of 11 milliseconds to generate 256 bits of entropy.

For the best-case scenario the assumption is made that the Android device is constantly in motion, and therefore all six decimals (including the first decimal that was predictable with a motionless device) contain entropy. Secondly, the assumption is made that each of these six decimals only affect the least significant bit after their IEEE 754 conversion. Under these assumptions, one sensor reading would provide 6 bits of entropy for the entropy pool. So in order to reach the 256-bit entropy threshold, the hash would need to be fed 43 sensor readings. Taking the average of 374600 sensor readings from table 3.2, this results in a duration of 7 milliseconds to generate 256 bits of entropy.

	Session 1	Session 2
Nexus 7, 60 seconds laying motionless on a desk	378800	376896
Nexus 7, 60 seconds making different movement motions	383249	379427
Nexus 4, 60 seconds laying motionless on a desk	369356	369595
Nexus 4, 60 seconds making different movement motions	369303	370175

Table 3.2: Total amount of sensor readings with Nexus 7 and Nexus 4 under different circumstances

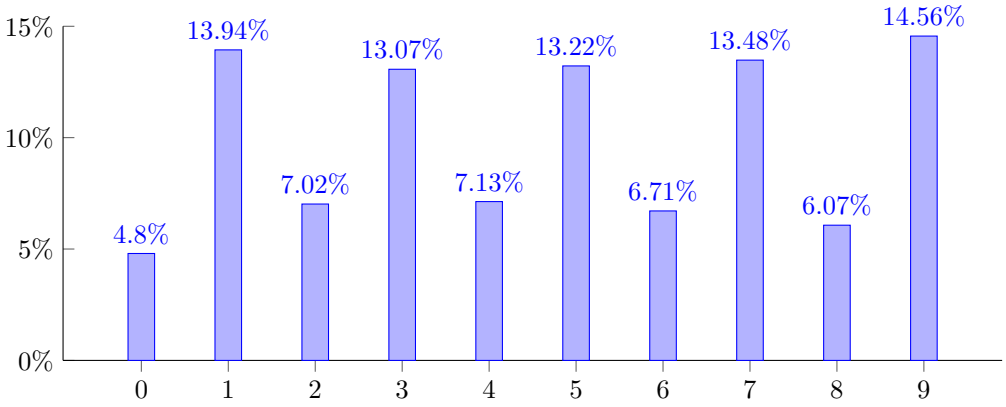

Figure 3.8: Histogram of the fifth decimal value occurrences

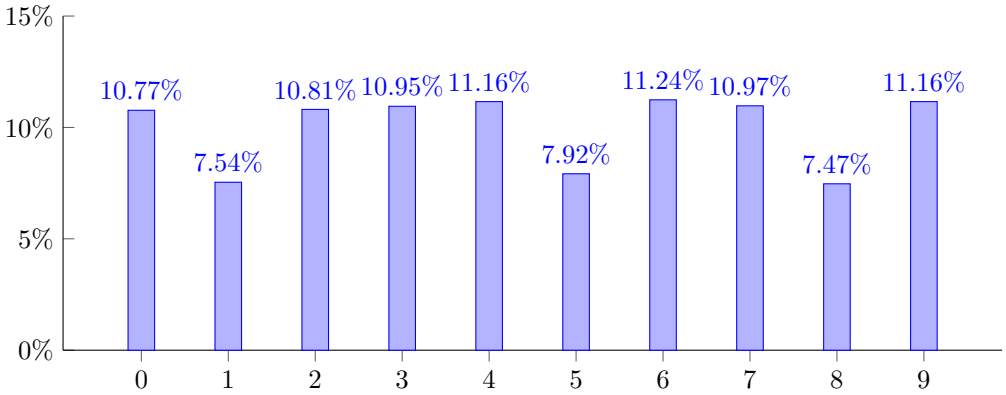

Figure 3.9: Histogram of the sixth decimal value occurrences

Taking the three different scenarios, the theoretical range of the entropy gathering duration will be anywhere between 7 and 42 milliseconds, because one sensor reading would provide anywhere between 1 and 6 bits of entropy. The paper by Voris et al. [40] shows a min-entropy of between 3.4 and 3.9 bits for a stationary accelerometer, so this is consistent with the 1 to 6 bits indication of the full sensor result scenarios.

Since the circumstances in an actual entropy gathering scenario in the field might still be very different from the ideal circumstances, the prototype also holds a very large margin of error to make sure the entropy threshold is met. The prototype will gather entropy from sensor data for a full 60 seconds before extracting the resulting randomness from the hash. Even when taking the worst-case scenario, this theoretically results in 374600 bits of entropy. This means that the actual entropy level of the sensor data can be 1463 times lower than the worst-case scenario and it will still meet the 256 bit security threshold. This margin of error should be more than enough to reach the 256-bit security threshold, even when circumstances in the field would differ significantly from the theoretical scenarios.

3.2 Performance Results

In this section I cover the results of the research. First, I summarize and explain the results from the DataGeneration app. Then I cover the prototype results and I make an assessment based on a number of criteria to show how the prototype performs and what additional security the prototype concept can provide.

3.2.1 Data generation results

To get a further indication of the entropy quality of the sensor data, large amounts of sensor data samples have been gathered using the DataGeneration app. The full source code of the DataGeneration app can be found in Appendix B.

Two sensor data samples have been collected with the DataGeneration app. The first sample used a Nexus 4 smart phone running Android OS 5.1.1, and the second sample used a Nexus 7 tablet running Android OS 5.1.1. During each sample run a total of 20 collection runs have been performed, which took a total of 5 minutes. During this time, sensor data has been collected from a number of sensors which were present on the devices. Google gives the following description of each of the present sensors in their API documentation [21]:

Rotation Vector
The rotation vector represents the orientation of the device as a combination of an *angle* and an *axis*, in which the device has rotated through an angle θ around axis $< x, y, z >$. The three coordinates are defined as a direct orthonormal basis, where:

- X is defined as the vector product **Y.Z** (It is tangential to the ground at the device's current location and roughly points East).
- Y is tangential to the ground at the device's current location and points towards magnetic north.
- Z points towards the sky and is perpendicular to the ground.

The resulting values from this sensor which are used in the entropy pool are:

- $values[0] : x * sin(\theta/2)$
- $values[1] : y * sin(\theta/2)$
- $values[2] : z * sin(\theta/2)$
- $values[3] : cos(\theta/2)$
- $values[4]$: estimated heading Accuracy (in radians).

Game Rotation Vector
Identical to the Rotation Vector, except that it does not use the geomagnetic field. Therefore the Y axis does not point north, but instead to some other reference, that reference is allowed to drift by the same order of magnitude as the gyroscope drift around the Z axis. Also, this sensor does not have an estimated heading accuracy value.

So, the resulting values from this sensor which are used in the entropy pool are similar to the Rotation Vector results, but without the fifth value for the estimated heading accuracy value.

Accelerometer
A sensor of this type measures the acceleration applied to the device (**Ad**). Conceptually, it does so by measuring forces applied to the sensor itself (**Fs**) using the relation:

$Ad = -g - \sum Fs/mass$, where g represents the force of gravity.
So $g = 9.81 m/s^2$ when the device is laying motionless on a desk.

The resulting values from this sensor which are used in the entropy pool are:

- $values[0]$: Acceleration minus Gx on the x-axis.
- $values[1]$: Acceleration minus Gy on the y-axis.
- $values[2]$: Acceleration minus Gz on the z-axis.

Linear Acceleration
A three dimensional vector indicating acceleration along each device axis, not including gravity. All values have units of m/s^2. The linear acceleration sensor output is correlated with the accelerometer and gravity sensor outputs in the following relation:

Acceleration = gravity + linear-acceleration

The resulting coordinate values are the same as used by the accelerometer.

Gravity
A three dimensional vector indicating the direction and magnitude of gravity. Units are m/s^2. The gravity sensor output is correlated with the accelerometer and linear acceleration sensor outputs in the following relation:

Acceleration = gravity + linear-acceleration

The resulting coordinate values are the same as used by the accelerometer.

Orientation
This sensor outputs the device orientation angles in degrees. The resulting values for use in the entropy pool are:

- $values[0]$: Azimuth, angle between the magnetic north direction and the y-axis, around the z-axis (0 to 359). 0=North, 90=East, 180=South, 270=West.
- $values[1]$: Pitch, rotation around x-axis (-180 to 180), with positive values when the z-axis moves **toward** the y-axis.
- $values[2]$: Roll, rotation around the y-axis (-90 to 90) increasing as the device moves clockwise.

Gyroscope
The gyroscope measures the rate of rotation around the device's local X, Y and Z axis and all values are in radians/second. The resulting values for the entropy pool are as follows:

- $values[0]$: Angular speed around the x-axis.
- $values[1]$: Angular speed around the y-axis.
- $values[2]$: Angular speed around the z-axis.

Light
This sensor measures the ambient light level, it outputs one value for use in the entropy pool:

$values[0]$: Ambient light level in SI lux units.

Proximity
The proximity sensor detects the presence of nearby objects, usually through an electromagnetic field like infrared. It outputs one value for use in the entropy pool:

$values[0]$: Proximity sensor distance measured in centimeters.

Barometer
The barometer measures the atmospheric pressure, it outputs one value for use in the entropy pool:

$values[0]$: Atmospheric pressure in hPa (millibar).

Magnetometer
The magnetometer measures the ambient magnetic field in the X, Y and Z axis. This results in three output values for use in the entropy pool:

- $values[0]$: Ambient magnetic field in the x-axis in μT (micro-Tesla).
- $values[1]$: Ambient magnetic field in the y-axis in μT (micro-Tesla).
- $values[2]$: Ambient magnetic field in the z-axis in μT (micro-Tesla).

For the first sample using the Nexus 4 smart phone, 13.548.859 bits of data have been collected. This data has been collected from 11 different sensors that were present on the device. Table 3.3 lists the occurrences of the value 1 and the value 0 for each of the 13.548.859 bits of collected data. These occurrences give an indication of the amount of entropy that these sensors provide, ideally the sensors would provide exactly the same amount of ones as zeros.

However, since the sensor data is converted to the IEEE 754 standard, it contains a margin of error because of the sign and exponent bits. The entropy of the sensor data is contained in the mantissa, which are the last 23 bits of the IEEE 754 notation. This means that for every result of 32 bits, the first 9 bits will contain little or no entropy. This gives an inherent margin of error of around 28%. Despite this margin of error, these results should still give an indication of the amount of entropy contained in the sensor data.

Table 3.3 shows that most sensors have a quite even distribution of ones and zeros in their output. Sensors like Proximity and Light perform poorly in the sample, but their sample size is also significantly smaller than the other sensors. Despite some sensors performing poorly, the total distribution still results in 45.89% ones and 54.11% zeros, which is a very good result given the 28% margin of error.

	#1s	% of total		#0s	% of total
Rotation Vector	1223596	55.73%		971800	44.27%
Game Rotation Vector	990246	56.78%		753760	43.22%
Accelerometer	524007	38.87%		823943	61.13%
Linear Acceleration	686440	50.82%		664329	49.18%
Gravity	618374	45.97%		726894	54.03%
Orientation	618401	45.48%		741468	54.52%
Magnetometer	385015	37.27%		648010	62.73%
Proximity	30	24%		95	76%
Light	1139	11.23%		8999	88.77%
Barometer	34826	51.25%		33126	48.75%
Gyroscope	1134974	36.68%		1959387	63.32%
Total	6217048	45.89%		7331811	54.11%

Table 3.3: Occurrences of the number of ones and zeros in the Nexus 4 sample

For the second sample using the Nexus 7 tablet, 7.764.197 bits of data have been collected. This data has been collected from 9 different sensors that were present on the device. Table 3.4 lists the occurrences of the value 1 and the value 0 for each of the 7.764.197 bits of collected data.

It is interesting to see that most sensors that the Nexus 4 and the Nexus 7 devices have in common perform almost equally in their distribution across the two devices. Another interesting result is that the Linear Acceleration sensor first performed the best of all sensors in the Nexus 4 sample, with an almost equal distribution. And in this sample on the Nexus 7 the Linear Acceleration sensor has performed even better with a near perfect distribution of 49.998% ones and 50.002% zeros. Across all sensors on both devices, the Linear Acceleration sensor appears to be the best performing sensor by quite a margin.

Again, despite some sensors performing poorly, the Nexus 7 sample also performs very well with a total distribution of 46.88% ones and 53.12% zeros. Given the 28% margin of error from the IEEE 754 conversion, this is a very good result and it indicates that this sample could also contain a decent amount of entropy.

	#1s	% of total		#0s	% of total
Rotation Vector	721745	54.88%		593289	45.12%
Game Rotation Vector	589814	56.94%		446122	43.06%
Accelerometer	296032	36.97%		504686	63.03%
Linear Acceleration	401661	49.998%		401691	50.002%
Gravity	369274	46.24%		429374	53.76%
Orientation	364245	45.10%		443416	54.90%
Magnetometer	225093	37.63%		373015	62.37%
Light	2903	36.33%		5088	63.67%
Gyroscope	668704	41.88%		928045	58.12%
Total	3639471	46.88%		4124726	53.12%

Table 3.4: Occurrences of the number of ones and zeros in the Nexus 7 sample

Across both samples, both devices give an almost equal distribution of ones and zeros. Some sensors like Light and Proximity perform poorly and also produce very little data, so it is likely that these sensors will not produce a lot of entropy. However, it is still good to add these results to the entropy pool for the extractor, because they could still contain some entropy in their data which will help strengthen the security of the RNG system. Since the data in the entropy pool is

extracted by the hash function, any data added to the entropy pool can never reduce the amount of entropy. So adding low-entropy data can never harm the security of the system, as long as there is sufficient entropy in the entropy pool as a whole once it is extracted by the hash. Overall, the sample results indicate that Android device sensor data could contain a high amount of entropy, and that the sensor data is fit for entropy generation for a random-number generator.

3.2.2 Prototype results

To assess the performance of the prototype in terms of the statistical randomness quality, a large amount of output from the prototype has been tested using the Diehard randomness testing battery [27]. The Diehard battery runs 16 different randomness testing methods. The documentation of the Diehard battery describes the different testing methods as follows [27]:

Birthday Spacings Test
Choose m birthdays in a year of n days. List the spacings between the birthdays.If j is the number of values that occur more than once in that list, then j is asymptotically Poisson distributed with mean $m^3/(4n)$. Experience shows n must be quite large, say $n \geq 2^{18}$, for comparing the results to the Poisson distribution with that mean. This test uses $n = 2^{24}$ and $m = 2^9$, so that the underlying distribution for j is taken to be Poisson with $\lambda = 2^{27}/(2^{26}) = 2$. A sample of 500 j's is taken, and a chi-square goodness of fit test provides a p-value. The first test uses bits 1-24 (counting from the left) from integers in the specified file. Then the file is closed and reopened. Next, bits 2-25 are used to provide birthdays, then 3-26 and so on to bits 9-32. Each set of bits provides a p-value, and the nine p-values provide a sample for a KSTEST.

The Overlapping 5-permutation test
The OPERM5 test looks at a sequence of one million 32-bit random integers. Each set of five consecutive integers can be in one of 120 states, for the 5! possible orderings of five numbers. Thus the 5th, 6th, 7th,...numbers each provide a state. As many thousands of state transitions are observed, cumulative counts are made of the number of occurrences of each state. Then the quadratic form in the weak inverse of the 120x120 covariance matrix yields a test equivalent to the likelihood ratio test that the 120 cell counts came from the specified (asymptotically) normal distribution with the specified 120x120 covariance matrix (with rank 99). This version uses 1,000,000 integers, twice.

Binary Rank Test for 31x31 matrices
The leftmost 31 bits of 31 random integers from the test sequence are used to form a 31x31 binary matrix over the field $0, 1$. The rank is determined. That rank can be from 0 to 31, but ranks < 28 are rare, and their counts are pooled with those for rank 28. Ranks are found for 40,000 such random matrices and a chi-square test is performed on counts for ranks 31,30,29 and ≤ 28.

Binary Rank Test for 32x32 matrices
A random 32x32 binary matrix is formed, each row a 32-bit random integer. The rank is determined. That rank can be from 0 to 32, ranks less than 29 are rare, and their counts are pooled with those for rank 29. Ranks are found for 40,000 such random matrices and a chi-square test is performed on counts for ranks 32,31,30 and ≤ 29.

Binary Rank Test for 6x8 matrices
From each of six random 32-bit integers from the generator under test, a specified byte is chosen, and the resulting six bytes form a 6x8 binary matrix whose rank is determined. That rank can be from 0 to 6, but ranks 0,1,2,3 are rare; their counts are pooled with those for rank 4. Ranks are found for 100,000 random matrices, and a chi-square test is performed on counts for ranks 6,5 and ≤ 4.

Bitstream Test
The file under test is viewed as a stream of bits. Call them b1,b2,... . Consider an alphabet with two "letters", 0 and 1 and think of the stream of bits as a succession of 20-letter "words", overlapping. Thus the first word is b1b2...b20, the second is b2b3...b21, and so on. The bitstream test counts the number of missing 20-letter (20-bit) words in a string of 2^{21} overlapping 20-letter words. There are 2^{20} possible 20 letter words. For a truly random string of $2^{21}+19$ bits, the number of missing words j should be (very close to) normally distributed with mean 141,909 and $\sigma = 428$. Thus $(j - 141909)/428$ should be a standard normal variate (z score) that leads to a uniform $[0, 1)$ p-value. The test is repeated twenty times.

OPSO, OQSO and DNA Tests
OPSO means Overlapping-Pairs-Sparse-Occupancy. The OPSO test considers 2-letter words from an alphabet of 1024 letters. Each letter is determined by a specified ten bits from a 32-bit integer in the sequence to be tested. OPSO generates 2^{21} (overlapping) 2-letter words (from $2^{21}+1$ "keystrokes") and counts the number of missing words—that is 2-letter words which do not appear in the entire sequence. That count should be very close to normally distributed with mean 141,909, $\sigma = 290$. Thus (missingwrds-141909)/290 should be a standard normal variable. The OPSO test takes 32 bits at a time from the test file and uses a designated set of ten consecutive bits. It then restarts the file for the next designated 10 bits, and so on.

OQSO means Overlapping-Quadruples-Sparse-Occupancy. The test OQSO is similar, except that it considers 4-letter words from an alphabet of 32 letters, each letter determined by a designated string of 5 consecutive bits from the test file, elements of which are assumed 32-bit random integers. The mean number of missing words in a sequence of 2^{21} four-letter words, $(2^{21}+3$ "keystrokes"), is again 141909, with $\sigma = 295$. The mean is based on theory; σ comes from extensive simulation.

The DNA test considers an alphabet of 4 letters: C,G,A,T, determined by two designated bits in the sequence of random integers being tested. It considers 10-letter words, so that as in OPSO and OQSO, there are 2^{20} possible words, and the mean number of missing words from a string of 2^{21} (overlapping) 10-letter words ($2^{21}+9$ "keystrokes") is 141909. The standard deviation $\sigma = 339$ was determined as for OQSO by simulation. (σ for OPSO, 290, is the true value (to three places), not determined by simulation.)

Count-the-1's Test (stream of bytes)
Consider the file under test as a stream of bytes (four per 32 bit integer). Each byte can contain from 0 to 8 1's, with probabilities 1,8,28,56,70,56,28,8,1 over 256. Now let the stream of bytes provide a string of overlapping 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's in a byte: 0,1,or 2 yield A, 3 yields B, 4 yields C, 5 yields D and 6,7 or 8 yield E. Thus we have a monkey at a typewriter hitting five keys with various probabilities (37,56,70,56,37 over 256). There are 5^5 possible 5-letter words, and from a string of 256,000 (overlapping) 5-letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chi-square test: Q5-Q4, the difference of the naive Pearson sums of $(\text{OBS-EXP})^2/\text{EXP}$ on counts for 5- and 4-letter cell counts.

Count-the-1's Test (specific bytes)
Consider the file under test as a stream of 32-bit integers. From each integer, a specific byte is chosen , say the left-most: bits 1 to 8. Each byte can contain from 0 to 8 1's, with probabilities 1,8,28,56,70,56,28,8,1 over 256. Now let the specified bytes from successive integers provide a string of (overlapping) 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's, in that byte: 0,1,or 2 —> A, 3 —> B, 4 —> C, 5 —> D, and 6,7 or 8 —> E. Thus we have a monkey at a typewriter hitting five keys with with various probabilities: 37,56,70,56,37 over 256. There are 5^5 possible 5-letter words, and from a string of 256,000 (overlapping) 5-letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chi-square test: Q5-Q4, the difference of the naive Pearson sums of (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.

Parking Lot Test
In a square of side 100, randomly "park" a car—a circle of radius 1. Then try to park a 2nd, a 3rd, and so on, each time parking "by ear". That is, if an attempt to park a car causes a crash with one already parked, try again at a new random location. (To avoid path problems, consider parking helicopters rather than cars.) Each attempt leads to either a crash or a success, the latter followed by an increment to the list of cars already parked. If we plot n: the number of attempts, versus k: the number successfully parked, we get a curve that should be similar to those provided by a perfect random-number generator. Theory for the behavior of such a random curve seems beyond reach, and as graphics displays are not available for this battery of tests, a simple characterization of the random experiment is used: k, the number of cars successfully parked after $n = 12,000$ attempts. Simulation shows that k should average 3523 with $\sigma = 21.9$ and is very close to normally distributed. Thus $(k - 3523)/21.9$ should be a standard normal variable, which, converted to a uniform variable, provides input to a KSTEST based on a sample of 10.

Minimum Distance Test
It does this 100 times: choose n=8000 random points in a square of side 10000. Find d, the minimum distance between the $(n^2-n)/2$ pairs of points. If the points are truly independent uniform, then d^2, the square of the minimum distance should be (very close to) exponentially distributed with mean .995 . Thus 1-exp($-d^2/.995$) should be uniform on $[0, 1)$ and a KSTEST on the resulting 100 values serves as a test of uniformity for random points in the square. Test numbers=0 mod 5 are printed but the KSTEST is based on the full set of 100 random choices of 8000 points in the 10000x10000 square.

3D Spheres Test
Choose 4000 random points in a cube of edge 1000. At each point, center a sphere large enough to reach the next closest point. Then the volume of the smallest such sphere is (very close to) exponentially distributed with mean $120\pi/3$. Thus the radius cubed is exponential with mean 30. (The mean is obtained by extensive simulation). The 3DSPHERES test generates 4000 such spheres 20 times. Each min radius cubed leads to a uniform variable by means of 1-exp($-r^3/30.$), then a KSTEST is done on the 20 p-values.

Squeeze Test
Random integers are floated to get uniforms on $[0, 1)$. Starting with $k = 2^{31} = 2147483647$, the test finds j, the number of iterations necessary to reduce k to 1, using the reduction $k = \lceil (k * U) \rceil$, with U provided by floating integers from the file being tested. Such j's are found 100,000 times, then counts for the number of times j was $\leq 6, 7, ..., 47, \geq 48$ are used to provide a chi-square test for cell frequencies.

Overlapping Sums Test
Integers are floated to get a sequence $U(1), U(2), ...$ of uniform $[0, 1)$ variables. Then overlapping sums, $S(1) = U(1) + ... + U(100), S2 = U(2) + ... + U(101), ...$ are formed. The S's are virtually normal with a certain covariance matrix. A linear transformation of the S's converts them to a sequence of independent standard normals, which are converted to uniform variables for a KSTEST. The p-values from ten KSTESTs are given still another KSTEST.

Runs Test
This test counts runs up, and runs down, in a sequence of uniform $[0, 1)$ variables, obtained by floating the 32-bit integers in the specified file. This example shows how runs are counted: .123,.357,.789,.425,.224,.416,.95 contains an up-run of length 3, a down-run of length 2 and an up-run of (at least) 2, depending on the next values. The covariance matrices for the runs-up and runs-down are well known, leading to chi-square tests for quadratic forms in the weak inverses of the covariance matrices. Runs are counted for sequences of length 10,000. This is done ten times. Then repeated.

Craps Test
This test plays 200,000 games of craps, finds the number of wins and the number of throws necessary to end each game. The number of wins should be (very close to) a normal with mean 200000p and variance 200000p(1-p), with p=244/495. Throws necessary to complete the game can vary from 1 to infinity, but counts for all>21 are lumped with 21. A chi-square test is made on the no.-of-throws cell counts. Each 32-bit integer from the test file provides the value for the throw of a die, by floating to $[0, 1)$, multiplying by 6 and taking 1 plus the integer part of the result.

To test the sensor data prototype RNG, the Diehard battery has been given 10 megabytes of randomness output from the prototype to process. This output is then tested by the 16 testing methods which assign p-values to the processed data, a total of 233 p-values are assigned by the 16 testing methods. These p-values represent the statistical distribution of the data. A RNG system has passed the Diehard tests if the resulting p-values are uniformly distributed over the field $0, 1$. A digression from the uniform distribution indicates that some Diehard tests have detected predictable patterns in the data.

The graph in Figure 3.10 shows the distribution of the prototype output in blue, and the perfect uniform distribution as point of reference in red. The y-axis represents the p-values and the x-axis represents the sorted distribution of all 233 resulting p-values from lowest to highest. It can be seen in the graph that the prototype output passed the Diehard tests, since its distribution is very close to the perfect uniform distribution. It is to be expected that the output is statistically random after extraction by the XSalsa20 PRNG, but it serves as a validation that the prototype construction model is securely implemented.

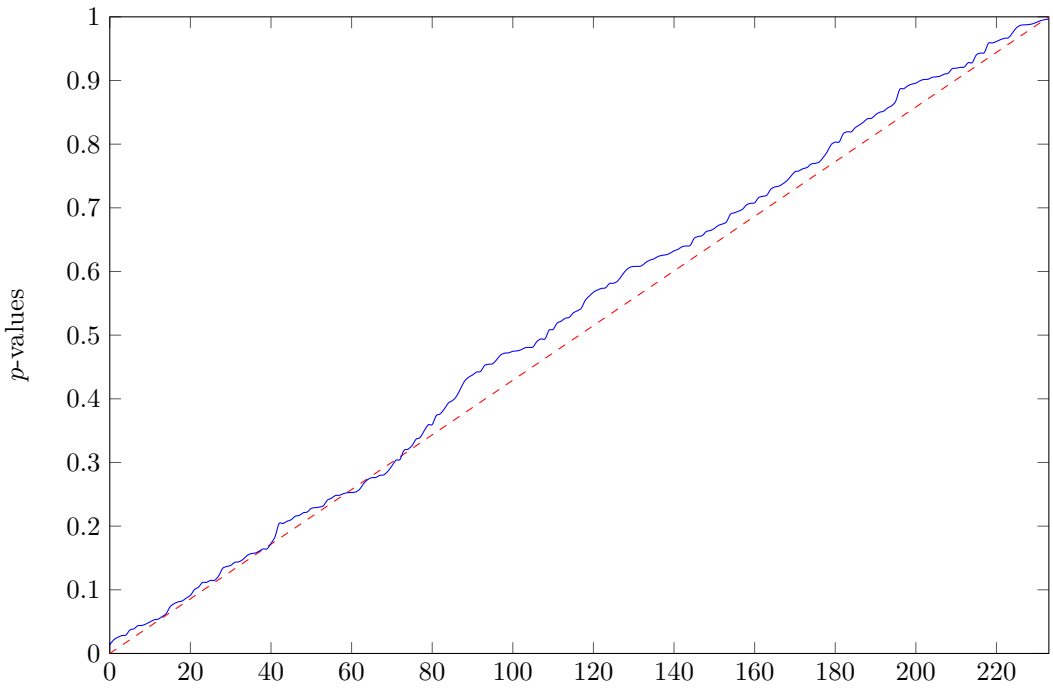

Figure 3.10: Sorted distribution of the Diehard test results on prototype randomness output

The performance of the prototype service process on the Android device is also considered for the overall quality. The daemon should not bother the device user by slowing down or even blocking the device. So it is important that the entropy generation process happens efficiently and without blocking the user from performing his normal activities on the device. The prototype performance can be assessed by using the following criteria:

- Duration of entropy pool generation
- CPU resources used
- RAM resources used
- Entropy quality

Duration of entropy pool generation
Currently set to 60 seconds, but this is set to a very conservative value to give the highest guarantee of security. This can likely be set as low as 5-10 seconds without dropping below the entropy threshold of 256 bits.

CPU resources used
While generating the entropy pool the CPU usage can spike quite high. But after the entropy pool has been filled and the daemon runs stationary, the CPU usage drops to almost 0%. When randomness is requested CPU usage goes up slightly, but this should not bother the user.

RAM resources used
Uses a high amount of RAM during entropy pool generation, for security reasons the entire entropy pool is stored in the RAM. But after the entropy generation is done, the daemon uses limited RAM resources.

Entropy quality
Entropy estimation testing shows that the prototype can gather the required 256 bits of entropy within as low as 7 to 42 milliseconds. Further testing shows that the data from most sensors is of very high entropy quality. Testing the RNG output with the Diehard tests verifies that the RNG output is statistically random.

The performance on these criteria show that the daemon will not bother the user by slowing down the device, since it only uses limited resources in terms of CPU and RAM once it is running stationary. The daemon will use more device resources during entropy pool generation, but this is only needed once during device boot-up for a limited duration.

Overall, the prototype has created a solid concept which can be used to provide additional security on Android devices. The results of the different statistical tests and analyses show that the prototype performs very well on the entropy and statistical randomness levels, which should contribute to strengthening the entropy pool of `/dev/random` when used in conjunction. The results of this thesis and research on the accelerometer by Voris et al. [40] show that sensor data contains a high amount of entropy even when the device is stationary. This means that sensor data does not require input from the user to provide high entropy, which helps to solve the vulnerability of `/dev/random` which needs some user input to reach its entropy threshold. Entropy generation from sensor data also has a good perspective for the future, because smart devices are rapidly developing with more complex sensors which will only increase the entropy from sensor data over time.

Chapter 4

Conclusion

The goal of this research was to provide a strong source of randomness which can be used in addition to randomness from existing sources like `/dev/random`. This new source of randomness should contribute to solving the vulnerabilities that have been shown in `/dev/random` in terms of low-entropy events. The entropy should not be dependent on user interaction, else it would create moments of low entropy which can compromise the entropy pool. For this purpose I have suggested using device sensor data to obtain high-entropy data, which can be used in conjunction with `/dev/random` to strengthen the entropy pool and reduce the amount of low-entropy events as described by Dodis et al. [16].

In this research I have shown that it is possible to create a random-number generator using entropy gathered from Android device sensor data. The results have indicated that sensor data samples contain a high amount of unpredictability, especially in certain decimals of their readings. Especially the fourth decimal contains a very even distribution of values, which could indicate that at least the least significant bits of the sensor readings are suitable for entropy extraction. Tests have also shown that there are an average of 374600 sensor readings per minute, which means that the 256-bit security threshold can be achieved within as low as 7 to 42 milliseconds after boot-up.

Further research has demonstrated that the distribution of the number of ones and zeros in the sensor readings is very equal in most sensors, despite the large margin of error in the bit conversion. Especially the Linear Acceleration sensor has shown great promise for entropy generation with an almost perfect distribution of 49.998% ones and 50.002% zeros in the Nexus 7 sample. But, even the sensors with a lower distribution can still contribute a good amount of entropy to the entropy pool over a longer period of data generation. So overall the entropy pool should be of a high quality and fit for use in cryptographic protocols after extraction, assuming that the sensors are not heavily correlated.

An assessment of the randomness quality of the prototype output using the Diehard battery of randomness testing has demonstrated that the output has an almost perfect uniform distribution. Therefore, the randomness produced by the prototype is qualified to perform in cryptographic settings like generating keys for encryption schemes or generating nonces for authentication protocols.

The prototype has also shown to have a high performance after the initial process of filling the entropy pool. After the pool is filled and the random-number generator can run stationary, it uses a limited amount of CPU power and RAM storage space. Therefore, the user is not bothered in their normal device activities when the daemon is running in the background.

Overall, it can be concluded that using sensor data is a feasible way to efficiently extract a high amount of entropy, which does not require user interaction. With some adaptations the prototype can be adapted to work in an operational environment, preferably in conjunction with

`/dev/random` to provide a maximum level of security. Combining the strengths of the high entropy in sensor data with the existing foundation of low level hardware entropy in `/dev/random` will help protect Android device users from security vulnerabilities related to entropy weaknesses in the future. However, as shown in the motivation in Section 1.2, it is often the human factor that fails in security incidents. Despite having encryption keys of the highest level of randomness, it can still be compromised by the human factor in a security setting.

4.1 Future Work

The research field of randomness and entropy generation is still quite an unexplored field of research in some aspects, especially on smart devices. The research field will be a constant arms race between attackers and developers of cryptographic systems. There are a number of different aspects that still need further research in this area:

- **Build the same concept for Apple devices**

  This thesis has fully dedicated research on Android devices, similar research could be done on Apple devices to see if they can also provide a high level of entropy from their sensors. The entropy gathered from the sensors could then be compared with the platform defaults for randomness generation and perhaps also form a strong conjunction by combining the sensor data entropy with entropy from existing standard algorithms.

- **Create an adapted version of the prototype for use in the field**

  Currently, the prototype is only built for demonstration and research purposes. For future research, the prototype could be adapted to create a fully operational version which can be released in the field through app stores. The prototype would have to be adapted into an integrated version which automatically combines the sensor data with `/dev/random` to create a fully functional method for large entropy requests. It would also need to work with intentFilters instead of running as a localService, so it can receive requests from other apps. An example of a usage scenario would be a mobile banking app which requests randomness from the adapted RNG for use in the encrypted communication and authentication with the bank servers.

- **Create a dynamic sensor data generator which adapts to user behaviour**

  The prototype currently generates sensor data for its entropy pool once on device boot-up. After the entropy pool has been filled, it will not refresh the stream cipher key until the next boot-up. The prototype could be adapted to monitor user behaviour based on touch pad signals or device movement. During times without user interaction (e.g., a phone sitting idle in a user's pocket), the prototype could start gathering new sensor data for the entropy pool to refresh the keys. Since the average smart device user will probably have the device idle for large amounts of time, this could greatly increase the security of the RNG system.

- **Research the differences in sensor output between user- and kernel-space**

  Currently, the prototype runs as a service in the Android user-space. Research could be done to see if different sensor results with higher entropy can be obtained from the kernel-space. A process in the kernel-space could perhaps even gain access to sensors which are unavailable to the user-space. This would reduce user interaction and transparency, but it might increase entropy values if the sensor data in the kernel-space is more unpredictable.

- **Further research on entropy estimation**

  Making a good estimation of entropy is a difficult problem, it faces the paradox of not being able to 'predict' unpredictability. Even though this thesis has given a strong indication of entropy, a more precise estimation of the amount of entropy contained in the sensor data could greatly reduce the duration of the entropy generation process. This will make the random-number generator more efficient and it will bother the device user less with CPU and RAM usage.

Bibliography

[1] Charles Arthur. Apps vulnerable to hacking, warns security company, 2013. http://www.theguardian.com/technology/2013/dec/11/app-hacking-commonplace-warns-security-company, Accessed July 2015.

[2] Boaz Barak and Shai 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*, pages 203–212. ACM, 2005. https://eprint.iacr.org/2005/029.pdf.

[3] Boaz Barak, Russell Impagliazzo, and Avi Wigderson. Extracting randomness using few independent sources. *SIAM Journal on Computing*, 36(4):1095–1118, 2006. http://www.math.ias.edu/~avi/PUBLICATIONS/MYPAPERS/BIW04/BIW.pdf.

[4] Elaine Barker and John Kelsey. Recommendation for the entropy sources used for random bit generation. *Draft NIST Special Publication*, 2012. http://www.gocs.de/pages/kryptologie/archiv/SP800-90b.pdf.

[5] Luciano Bello and Maximiliano Bertacchini. Predictable PRNG in the vulnerable Debian OpenSSL package: the what and the how. 2008. http://www.researchgate.net/profile/Luciano_Bello/publication/255569058_Predictable_PRNG_In_The_Vulnerable_Debian_OpenSSL_Package/links/542d3a370cf29bbc126d2223.pdf, Accessed July 2015.

[6] Daniel J Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*, volume 8, 2008. http://cr.yp.to/chacha/chacha-20080128.pdf.

[7] Daniel J Bernstein. The Salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer, 2008. http://cr.yp.to/snuffle/salsafamily-20071225.pdf.

[8] Daniel J Bernstein. Entropy Attacks! *The cr.yp.to blog*, 2014. http://blog.cr.yp.to/20140205-entropy.html, Accessed August 2015.

[9] Daniel J Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko Van Someren. Factoring RSA keys from certified smart cards: Coppersmith in the wild. In *Advances in Cryptology-ASIACRYPT 2013*, pages 341–360. Springer, 2013.

[10] Bitcoin. Android security vulnerability, 2013. https://bitcoin.org/en/alert/2013-08-11-android, Accessed July 2015.

[11] Bushing, Marcan, Segher, Sven. Console Hacking 2010 - PS3 Epic Fail. 2010. https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf.

[12] Bouncy Castle. Bouncy castle crypto APIs. 2007. http://www.bouncycastle.org/, Accessed August 2015.

[13] Ben Cox. Auditing GitHub users SSH key quality, 2015. https://blog.benjojo.co.uk/post/auditing-github-users-keys, Accessed July 2015.

[14] Debian. DSA-1571-1 openssl – predictable random number generator, 2008. http://www.debian.org/security/2008/dsa-1571, Accessed July 2015.

[15] Prelov V.V. Dobrushin, R.L. Entropy. In *Encyclopedia of Mathematics*. Springer, 2001. http://www.encyclopediaofmath.org/index.php?title=Entropy&oldid=15099.

[16] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input:/dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 647–658. ACM, 2013.

[17] Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. How to eat your entropy and have it too–optimal recovery strategies for compromised rngs. In *Advances in Cryptology–CRYPTO 2014*, pages 37–54. Springer, 2014. https://secure.netsolhost.com/cryptome.org/2014/04/eat-your-entropy.pdf.

[18] eMarketer. 2 billion consumers worldwide to get smart(phones) by 2016, 2014. http://www.emarketer.com/Article/2-Billion-Consumers-Worldwide-Smartphones-by-2016/1011694, Accessed July 2015.

[19] Niels Ferguson and Bruce Schneier. *Practical cryptography*, volume 23. Wiley New York, 2003.

[20] Google. Android Developers: Dashboards, 2015. http://developer.android.com/about/dashboards/index.html, Accessed July 2015.

[21] Google. Android Developers: SensorEvent, 2015. http://developer.android.com/reference/android/hardware/SensorEvent.html, Accessed July 2015.

[22] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security Symposium*, pages 205–220, 2012. https://factorable.net/weakkeys12.extended.pdf.

[23] Intel. *Intel Digital Random Number Generator (DRNG)*, 2012. https://software.intel.com/sites/default/files/m/d/4/1/d/8/441_Intel_R__DRNG_Software_Implementation_Guide_final_Aug7.pdf, Accessed July 2015.

[24] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography*, pages 13–33. Springer, 2000.

[25] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption*, pages 168–188. Springer, 1998. https://www.schneier.com/paper-prngs.pdf.

[26] Andrei N Kolmogorov. On tables of random numbers. *Sankhyā: The Indian Journal of Statistics, Series A*, pages 369–376, 1963.

[27] George Marsaglia. Diehard: A battery of tests of randomness. 1996. http://www.stat.fsu.edu/pub/diehard/, Accessed August 2015.

[28] Jeff Larson Nicole Perlroth and Scott Shane. N.S.A. able to foil basic safeguards of privacy on web, 2013. http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html, Accessed August 2015.

[29] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[30] Benjamin Pousse. Short communication: An interpretation of the linux entropy estimator. *IACR Cryptology ePrint Archive*, 2012:487, 2012. http://eprint.iacr.org/2012/487.pdf.

[31] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, DTIC Document, 2001. http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html, Accessed August 2015.

[32] Nils Schneider. Recovering bitcoin private keys using weak signatures from the blockchain, 2013. http://www.nilsschneider.net/2013/01/28/recovering-bitcoin-private-keys.html.

[33] Bruce Schneier. Random number bug in debian linux, 2008. https://www.schneier.com/blog/archives/2008/05/random_number_b.html.

[34] SpongyCastle. Repackage of Bouncy Castle for Android. *Bouncy Castle Project*, 2012. https://rtyley.github.io/spongycastle/, Accessed August 2015.

[35] Theodore Ts'o. [PATCH -v4] random: introduce getrandom(2) system call. 2014. http://article.gmane.org/gmane.linux.kernel.cryptoapi/11733, Accessed August 2015.

[36] Theodore Ts'o and Matt Mackall. *random.c – A strong random number generator*. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/drivers/char/random.c, Accessed August 2015.

[37] US-CERT/NIST. Vulnerability Summary for CVE-2008-0166, 2008. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0166, Accessed July 2015.

[38] HCA van Tilborg. *Fundamentals of Cryptology*. Kluwer Academic Publishers, 1999.

[39] John Von Neumann. 13. various techniques used in connection with random digits. 1951. https://dornsifecms.usc.edu/assets/sites/520/docs/VonNeumann-ams12p36-38.pdf, Accessed July 2015.

[40] Jonathan Voris, Nitesh Saxena, and Tzipora Halevi. Accelerometers and randomness: perfect together. In *Proceedings of the fourth ACM conference on Wireless network security*, pages 115–126. ACM, 2011. https://cis.uab.edu/saxena/docs/wisec.pdf.

[41] Brian Warner. EGD: The Entropy Gathering Daemon. 2002. http://egd.sourceforge.net/, Accessed August 2015.

[42] Thomas Watson. The complexity of estimating min-entropy. *Computational Complexity*, pages 1–23, 2015. http://www.cs.toronto.edu/~thomasw/papers/min.pdf.

Appendix A

RandomnessGenerator Prototype

A.1 Technical Documentation

Technical documentation on the Randomness Generator MainService daemon and the Randomness Generator MainActivity user interface for the prototype demonstration to the user. The full source code of this prototype can also be found in sections A.2 and A.3.

Prototype specifications:

- **The prototype has been developed on the 3rd of July 2015.**
- **The prototype has been developed using Android Studio version 1.2.2.**
- **The prototype has an API level of 17, meaning that it supports Android OS version 4.2 and above.**
- **The prototype consists of the following components:**
  - **The MainService component which contains the background daemon which generates an entropy pool and takes requests for randomness after the entropy pool is of sufficient strength.**
  - **The MainActivity component which contains the user interface used for demonstration purposes of the background daemon. The user can use this user interface to request a certain amount of bits of randomness from the background daemon which will then be displayed on the screen in hexadecimal format to the user.**

Prototype installation guide:

The prototype can be installed using two different methods:

A) The package can be directly installed on an Android device using the RandomnessGenerator.apk file:

1. First make sure the device can install apps outside of Google Play by going to **Settings** → **Security** and enabling **"Unknown sources - Allow installation of apps from unknown sources"**.
2. Then simply start the RandomnessGenerator.apk file to install the app on the device, if the apk file was downloaded it can be found in the Downloads app.
3. The device should now have the package installed and it can be started by clicking the **Randomness Generator** app in the app list of the device.

B) Alternatively, the entire Android Studio project can be used to install the package from a PC which is found in the RandomnessGenerator.7z file:

1. Download and install Android Studio on a PC from `https://developer.android.com/sdk/`
2. Unpack the RandomnessGenerator.7z to a location on the PC disk.
3. Start Android Studio and go to **File** → **Open...** and navigate to the extracted project folder, Android Studio will automatically recognize the folder as an Android Studio project and initialize it.
4. Connect the Android device to the PC.
5. Select **Run** → **Run 'app'** and choose the Android device from the list.
6. The Android device should now have the package installed and it will automatically run the user interface on the device.

User interface guide:

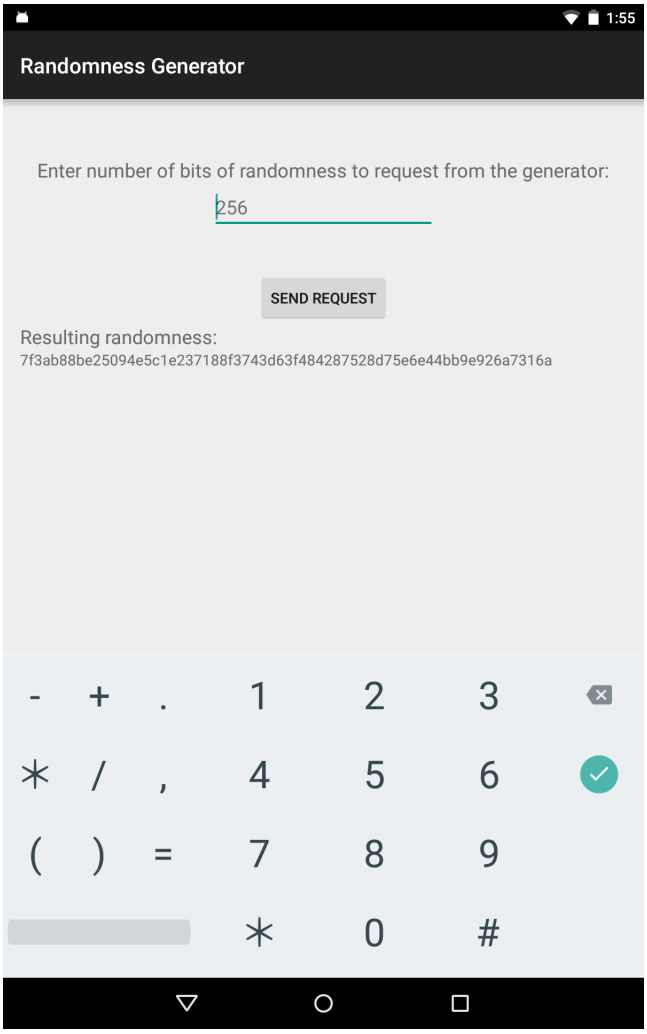

Figure A.1: Prototype user interface

Figure A.1 shows the user interface of the MainActivity of the prototype. This user interface is solely intended to demonstrate the communication between a cryptographic process and the randomness generator daemon. The user interface would not be a part of the package if the randomness generator daemon would be adapted for use in an actual setting in the field. In that case, all of the communication demonstrated in the user interface would take place in the background because it contains security sensitive data.

The user can interact with the user interface as follows:

- The user can use the text field at the top to enter a number of bits to request from the randomness generator daemon.
- The user can then press the **Send Request** button to initiate the communication to the service and request the entered number of bits of randomness.
- If the **Send Request** button is pressed by the user without entering a number of bits to request, the app will send a default request of 256 bits.
- When the request has been sent to the service, the service will send back a result which will be shown at the bottom section of the user interface.
- If the service has not been started yet at the time of the request or the service is not yet done generating a sufficient size entropy pool, an error message will also be shown to the user in the bottom section. In this case the user can simply press the button again a few seconds later to retry the request.

A.2 RandomnessGenerator MainService

Source code for the RandomnessGenerator MainService daemon which gathers entropy from the Android device sensor data, feeds this data to a SHA-256 hash and then uses the hash output as a key for the XSalsa20 algorithm to generate randomness on request.

```
package nl.ru.bcm.randomnessgenerator;

import android.app.Notification;
import android.app.Service;
import android.content.Intent;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.hardware.Sensor;
import android.os.Binder;
import android.os.IBinder;
import android.support.v4.app.NotificationCompat;

import org.spongycastle.crypto.StreamCipher;
import org.spongycastle.crypto.engines.XSalsa20Engine;
import org.spongycastle.crypto.params.KeyParameter;
import org.spongycastle.crypto.params.ParametersWithIV;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.Security;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class MainService extends Service implements SensorEventListener {
    private final IBinder mBinder = new LocalBinder();
```

```
    private boolean generatingEntropyPool;
    private long startTime, entropyPoolSize, entropyGatheringDuration = 60;
    private SensorManager mSensorManager;
    private static MessageDigest sha256 = null;
    private byte[] hash = null, IV = null;
    private StreamCipher cipher = null;
    private KeyParameter keyParams = null;
    private ParametersWithIV ivParams = null;


    //Service is created and initialization is performed
    @Override
    public void onCreate() {
        super.onCreate();
        //Build the notification icon to be displayed to the user
        NotificationCompat.Builder mBuilder =
                new NotificationCompat.Builder(this)
                        .setSmallIcon(R.drawable.emo_im_happy)
                        .setContentTitle("RandomnessGenerator")
                        .setContentText("Randomness Generator Daemon");
        Notification notification = mBuilder.build();

        //Set as a foreground service so the scheduler does not shut this process
            down after short periods of inactivity
        this.startForeground(444, notification);

        //Initialize flags, hash, sensors and the XSalsa20 cipher
        generatingEntropyPool = true;
        setHash();
        initSensors();
        Security.insertProviderAt(new org.spongycastle.jce.provider.
            BouncyCastleProvider(), 1);
        cipher = new XSalsa20Engine();

        //Start counting how long the entropy pool is gathering from sensors
        startTime = System.nanoTime();

        //Start counting amount of bytes of randomness added to the entropy pool
        entropyPoolSize = 0;
    }

    //Service is destroyed and all sensitive cryptographic elements are erased
    @Override
    public void onDestroy() {
        IV = null;
        hash = null;
        sha256 = null;
        cipher = null;
        keyParams = null;
        ivParams = null;
        entropyPoolSize = 0;
    }

    /*Service is started, this calls the onCreate method if the service was not
        already running
     * The activity can use this to create the service if it somehow has not
         already been started on phone boot-up
     */
    @Override
    public int onStartCommand(Intent intent, int flag, int id) {
        super.onStartCommand(intent, flag, id);
        return START_STICKY;
    }

    //Service is bound and returns the IBinder object
    @Override
    public IBinder onBind(Intent intent) {
```

```
        return mBinder;
    }

    public class LocalBinder extends Binder {
        public MainService getServiceInstance(){
            return MainService.this;
        }
    }

    /*Request a given amount of bits of randomness from the service
     * Returns a byte array with the requested amount of randomness rounded up to
         the nearest whole byte
     */
    public byte[] requestRandomness(int bits) {
        if(!generatingEntropyPool) {
            double dBits = (double) bits;
            double bytes = Math.ceil(dBits/8);
            byte[] zeroStream = new byte[(int)bytes];
            byte[] resultStream = new byte[(int)bytes];
            cipher.processBytes(zeroStream,0,zeroStream.length,resultStream,0);
            return resultStream;
        }
        else {
            return null;
        }
    }

    //Request amount of time remaining for the entropy pool generation process in
        seconds
    public long timeRemaining() {
        return (entropyGatheringDuration - TimeUnit.SECONDS.convert(System.nanoTime
            () - startTime, TimeUnit.NANOSECONDS));
    }

    //Initialize all available sensors and start listening
    private void initSensors(){
        mSensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
        List<Sensor> sensors = mSensorManager.getSensorList(Sensor.TYPE_ALL);
        for (Sensor sensor : sensors)
        {
            mSensorManager.registerListener(this, sensor, SensorManager.
                SENSOR_DELAY_FASTEST);
        }
    }

    //Close all sensor listeners
    private void closeSensors(){
        mSensorManager.unregisterListener(this);
    }

    /* Receives data from all sensors with registered listeners every sensor delay
        tick
     * Sensor data is fed to the hash
     * Once sufficient entropy has been generated the hash is extracted and the
         XSalsa20 algorithm is initialized
     */
    @Override
    public void onSensorChanged(SensorEvent event) {
        if(generatingEntropyPool) {
            for(int i = 0; i < event.values.length; i++) {
                if(Integer.toBinaryString(Float.floatToIntBits(event.values[i])).
                    length() != 1) {
                    int intData = Float.floatToIntBits(event.values[i]);
                    byte data[] = new byte[4];
                    data[0] = (byte) (intData >> 24);
                    data[1] = (byte) (intData >> 16);
                    data[2] = (byte) (intData >> 8);
```

```
                    data[3] = (byte) (intData);
                    updateHash(data);
                    //Update the entropy pool size
                    entropyPoolSize++;

                    /* Check if sufficient entropy has been generated for a strong
                        entropy pool
                     * This check is passed if n seconds have passed since the
                        start of the entropy generation where n >=
                        entropyGatheringDuration
                     * And if the entropy pool contains at least 2048 bits of
                        randomness
                     * Which is when entropyPoolSize >= 256 since entropyPoolSize
                        counts the number of bytes in the entropy pool
                     */
                    if((TimeUnit.SECONDS.convert(System.nanoTime() - startTime,
                        TimeUnit.NANOSECONDS) >= entropyGatheringDuration) && (
                        entropyPoolSize >= 256)) {
                        generatingEntropyPool = false;
                        hash = getHash();
                        keyParams = new KeyParameter(hash);
                        //Initiate the IV of the XSalsa20 cipher with 0
                        IV = new byte[24];
                        ivParams= new ParametersWithIV(keyParams,IV);
                        cipher.init(false,ivParams);
                        closeSensors();
                    }
                }
            }
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {

    }

    //Initialize the hash object as SHA-256
    private void setHash() {
        try {
            sha256 = MessageDigest.getInstance("SHA-256");
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        sha256.reset();
    }

    //Update the hash with new sensor data
    private void updateHash(byte[] data) {
        sha256.update(data);
    }

    //Digest the resulting 256 bit key from the hash
    private byte[] getHash() {
        return sha256.digest();
    }
}
```

A.3 RandomnessGenerator MainActivity

Source code for the RandomnessGenerator MainActivity app which was used in this thesis as a user interface to demonstrate the randomness generation of the RandomnessGenerator MainService background daemon and display the results to the user in a readable hexadecimal format.

```
package nl.ru.bcm.randomnessgenerator;

import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.support.v7.app.ActionBarActivity;
import android.text.TextUtils;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ScrollView;
import android.widget.TextView;

public class MainActivity extends ActionBarActivity {
    private Intent serviceIntent;
    private MainService mService;
    private Button start;
    private EditText mEditText;
    private TextView mTextView;
    private boolean serviceConnected = false;
    private byte[] result;

    //Activity is created and initialization is performed
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        //UI components are defined
        setContentView(R.layout.activity_main);
        start = (Button)findViewById(R.id.startButton);
        mEditText = (EditText)findViewById(R.id.editText);
        mTextView = new TextView(this);//(TextView)findViewById(R.id.resultText);
        mTextView.setTextAppearance(this,R.style.TextAppearance_AppCompat_Small);
        ScrollView mScrollView = (ScrollView)findViewById(R.id.scrollView);
        mScrollView.addView(mTextView);

        //Initialize the service daemon and bind to it
        serviceIntent = new Intent(MainActivity.this, MainService.class);
        startService(serviceIntent); //Starting the service
        bindService(serviceIntent, mConnection, Context.BIND_AUTO_CREATE); //
            Binding to the service!

        //Define behavior when 'Start' is pressed
        start.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                //Check if the service is connected
                if(serviceConnected) {
                    //Check if the user entered a number of bits in the entry field
                        , else default to a 256 bits request
                    if(TextUtils.isEmpty(mEditText.getText())) {
                        result = mService.requestRandomness(256);
                    }
                    else {
                        result = mService.requestRandomness(Integer.parseInt(
                            mEditText.getText().toString()));
                    }
                    //Check if the service returned a result and print the result
                        if so, else the service is still generating entropy and the
                         time remaining is printed
                    if (result != null) {
                        mTextView.setText(byteArrayToHexString(result));
                    }
                    else {
```

```
                        mTextView.setText("Service is still generating entropy pool
                            , please wait "+ mService.timeRemaining() +" seconds
                            before requesting randomness again.");
                    }
                }
                else {
                    mTextView.setText("Service is still initializing, please wait a
                         couple of seconds before requesting randomness again.");
                }
            }
        });

    }

    //When the user resumes the activity, check if the service is still connected,
        else rebind it
    @Override
    public void onResume() {
        super.onResume();
        if(!serviceConnected) {
            startService(serviceIntent); //Starting the service
            bindService(serviceIntent, mConnection, Context.BIND_AUTO_CREATE);
        }
    }

    //When the activity is stopped the service is unbound
    @Override
    public void onStop() {
        super.onStop();
        if(serviceConnected) {
            unbindService(mConnection);
            serviceConnected = false;
        }
    }

    //When the activity is destroyed the service is unbound
    @Override
    public void onDestroy() {
        super.onStop();
        if(serviceConnected) {
            unbindService(mConnection);
            serviceConnected = false;
        }
    }

    //Connection to the randomness generator service is initialized
    private ServiceConnection mConnection = new ServiceConnection() {

        @Override
        public void onServiceConnected(ComponentName className, IBinder service) {
            //LocalService is now bound, cast the IBinder and get LocalService
                instance
            MainService.LocalBinder binder = (MainService.LocalBinder) service;
            mService = binder.getServiceInstance(); //Get instance of the service
            serviceConnected = true;
        }

        @Override
        public void onServiceDisconnected(ComponentName arg0) {
            serviceConnected = false;
        }
    };

    //Transform the randomness from the received byte array to a readable
        hexadecimal string for the user
    private static String byteArrayToHexString(byte[] b) {
        String result = "";
```

```
        for (int i=0; i < b.length; i++) {
            result += Integer.toString( ( b[i] & 0xff ) + 0x100, 16).substring(1);
        }
        return result;
    }
}
```

Appendix B

DataGeneration app for sensor data generation

B.1 DataGeneration MainActivity

Source code for the DataGeneration MainActivity app which was used in this thesis to generate and store large amounts of sensor data for the statistical analysis of the entropy strength.

```
package nl.ru.thesis.DataGeneration;

import java.io.FileNotFoundException;
import java.util.ArrayList;

import android.os.Bundle;
import android.app.Activity;
import android.content.ClipboardManager;
import android.os.Environment;
import android.view.ContextMenu;
import android.view.ContextMenu.ContextMenuInfo;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.Spinner;
import android.widget.TextView;
import android.widget.Toast;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.hardware.Sensor;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.File;
import java.util.List;


public class MainActivity extends Activity implements SensorEventListener{
    //SETTINGS:
    //Maximum bytes of data stored in memory before writing data to storage:
    private final int maxArrayLength = 50000;
    //Maximum number of data collection runs the app will perform before closing: (
        Total size of gathered data on storage will be maxArrayLength *
        maxDataCollectionRuns)
    //Set this value to -1 to continue generating data indefinitely or until the
        user presses the Stop button
```

```
    private final int maxDataCollectionRuns = 20;

  private static final int ID_MENU_EXIT = 0;
  private SensorManager mSensorManager;
  private TextView text;
  private boolean read = false;
    private boolean started = false;
  private ArrayList<ArrayList<Integer>> valueList;
    private ArrayList<String> sensorNames;
    private int valueListSize = 0;
  private String source;
    private int timesWrittenToFile = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        //Create references to the layout
        final Button start = (Button) findViewById(R.id.button1);
        final Button stop = (Button) findViewById(R.id.button2);
        final Spinner spinner = (Spinner) findViewById(R.id.spinner1);
        text = (TextView) findViewById(R.id.textView3);

        //Register to allow copying
        registerForContextMenu(text);

       // Create an ArrayAdapter using the string array to add choices to the
           spinner
       ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
               R.array.choices, android.R.layout.simple_spinner_item);
       adapter.setDropDownViewResource(android.R.layout.
           simple_spinner_dropdown_item);
       spinner.setAdapter(adapter);



       //Define behavior when 'Start' is pressed
         start.setOnClickListener(new View.OnClickListener() {
             public void onClick(View v) {
               source = spinner.getSelectedItem().toString(); //Get source
                 System.out.println(source);
               Toast.makeText(MainActivity.this,source, Toast.LENGTH_SHORT).show();

               initSensors();
                 started = true;

           //initialize the data list and allow read
                 valueList = new ArrayList<ArrayList<Integer>>();
                 sensorNames = new ArrayList<String>();
                 if(source.compareTo("Accelerometer") == 0) {
                     sensorNames.add("AccelerometerX");
                     sensorNames.add("AccelerometerY");
                     sensorNames.add("AccelerometerZ");
                 }
           read = true;
           Toast.makeText(MainActivity.this,"Started", Toast.LENGTH_SHORT).show();

                 }
         });

       //Define behavior when 'Stop' is pressed - stop reading and print output
         stop.setOnClickListener(new View.OnClickListener() {
             public void onClick(View v) {
               stopApp();
             }
         });
```

```
    }

    private void stopApp() {
        read = false;
        if(started) {
            closeSensor();
            started = false;
        }
        text.setText("Stopped");
        System.out.println("Stopped");
    }
    /*
     * Initializes the device sensors by registering the appropriate Listeners for
         the chosen data generation method
     */
    private void initSensors(){
        mSensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
        String result = "";
        List<Sensor> sensors = mSensorManager.getSensorList(Sensor.TYPE_ALL);
        for (Sensor sensor : sensors)
        {
            result = result + sensor.getName() + ", ";
        }
        text.setText(result);

        if (source.compareTo("Full") == 0){
            for (Sensor sensor : sensors)
            {
                mSensorManager.registerListener(this, sensor, SensorManager.
                    SENSOR_DELAY_FASTEST);
            }
        }
      if (source.compareTo("Accelerometer") == 0)
        mSensorManager.registerListener(this, mSensorManager.getDefaultSensor(
            Sensor.TYPE_ACCELEROMETER), SensorManager.SENSOR_DELAY_FASTEST);
    }

    private void closeSensor(){
        mSensorManager.unregisterListener(this);
}

    //required but not used function for the menu
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        MenuItem item = menu.add(Menu.NONE,ID_MENU_EXIT,Menu.NONE,R.string.
            exitOption);

        return true;
    }

    //required but not used function for the accuracy of the sensor
@Override
public void onAccuracyChanged(Sensor arg0, int arg1) {

}

/**
 * Defines the behavior of sensor readings
 */
@Override
public void onSensorChanged(SensorEvent event) {
        //System.out.println("Sensor Changed");
    //If all arrays are 'full' stop reading and do further processing
    if (valueListSize >= maxArrayLength){
```

```
            read = false;
            PrintToFile();
        }

        if (read && source.compareTo("Full") == 0){
            //get the values of all the axis, up to 5 possible axis
            int addedValues = 0;
            boolean containedSensor = sensorNames.contains(event.sensor.getName());
            if(!containedSensor)
                sensorNames.add(event.sensor.getName());
            int index = sensorNames.indexOf(event.sensor.getName());
            ArrayList<Integer> dataList = new ArrayList<Integer>();
            //Check if not zero and array not full, than trim to suitable size
            if(valueListSize < maxArrayLength) {
                for(int i = 0; i < event.values.length; i++) {
                    if(Integer.toBinaryString(Float.floatToIntBits(event.values[i])
                        ).length() != 1) {
                        addedValues++;
                        dataList.add(Float.floatToIntBits(event.values[i]));
                    }
                }
            }
            if(addedValues == 0 && !containedSensor)
                sensorNames.remove(event.sensor.getName());
            if(addedValues > 0) {
                if (containedSensor) {
                    ArrayList<Integer> previousData = valueList.get(index);
                    previousData.addAll(dataList);
                    valueList.set(index, previousData);
                } else {
                    valueList.add(dataList);
                }
            }
            valueListSize += (addedValues*4);
        }
    if (event.sensor.getType()==Sensor.TYPE_ACCELEROMETER && read && source.
        compareTo("Accelerometer") == 0){
      //get the value of the three axis, x = event.values[0]), y = event.values[1],
          z = event.values[2]
            ArrayList<Integer> dataList0 = new ArrayList<Integer>();
            ArrayList<Integer> dataList1 = new ArrayList<Integer>();
            ArrayList<Integer> dataList2 = new ArrayList<Integer>();
            dataList0.add(Float.floatToIntBits(event.values[0]));
            dataList1.add(Float.floatToIntBits(event.values[1]));
            dataList2.add(Float.floatToIntBits(event.values[2]));
            int addedValues = 0;
      //Check if not zero and array not full, than trim to suitable size
      if (valueListSize  < maxArrayLength ) {
                if (valueList.isEmpty()) {
                    if (Integer.toBinaryString(Float.floatToIntBits(event.values
                        [0])).length() != 1) {
                        valueList.add(dataList0);
                        addedValues++;
                    }
                    else {
                        ArrayList<Integer> tempList = new ArrayList<Integer>();
                        valueList.add(tempList);
                    }
                    if (Integer.toBinaryString(Float.floatToIntBits(event.values
                        [1])).length() != 1) {
                        valueList.add(dataList1);
                        addedValues++;
                    }
                    else {
                        ArrayList<Integer> tempList = new ArrayList<Integer>();
                        valueList.add(tempList);
                    }
```

```
                    if (Integer.toBinaryString(Float.floatToIntBits(event.values
                        [2])).length() != 1) {
                        valueList.add(dataList2);
                        addedValues++;
                    }
                    else {
                        ArrayList<Integer> tempList = new ArrayList<Integer>();
                        valueList.add(tempList);
                    }
                }
                else {
                    if (Integer.toBinaryString(Float.floatToIntBits(event.values
                        [0])).length() != 1) {
                        ArrayList<Integer> previousData = valueList.get(0);
                        previousData.addAll(dataList0);
                        valueList.set(0, previousData);
                        addedValues++;
                    }
                    if (Integer.toBinaryString(Float.floatToIntBits(event.values
                        [1])).length() != 1) {
                        ArrayList<Integer> previousData = valueList.get(1);
                        previousData.addAll(dataList1);
                        valueList.set(1, previousData);
                        addedValues++;
                    }
                    if (Integer.toBinaryString(Float.floatToIntBits(event.values
                        [2])).length() != 1) {
                        ArrayList<Integer> previousData = valueList.get(2);
                        previousData.addAll(dataList2);
                        valueList.set(2, previousData);
                        addedValues++;
                    }
                }
                valueListSize += (addedValues*4);
            }
        }

}

    /*
     * Prints the resulting binary string to a file on the SD storage of the device
     */
    private void PrintToFile(){
        System.out.println("STARTING FILE WRITE");
        //Create a folder to write the results to
        File folder = new File(Environment.getExternalStorageDirectory() + "/
            results");
        if (!folder.exists()) {
            folder.mkdirs();
        }

        for (int i = 0; i < valueList.size(); i++){
            File file = new File(Environment.getExternalStorageDirectory() + "/
                results/" + sensorNames.get(i));
            //Create new file if it does not exist yet
            if (!(file.exists())) {
                try {
                    file.createNewFile();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("file created: " + file.getAbsolutePath());
            if (file.exists()) {
                OutputStream fo = null;
                try {
                    fo = new FileOutputStream(file, true);
```

```
                }
                catch (FileNotFoundException e) {
                    e.printStackTrace();
                }
                for (int j = 0; j < valueList.get(i).size(); j++) {
                    int intData = valueList.get(i).get(j);
                    byte data[] = new byte[4];
                    data[0] = (byte) (intData >> 24);
                    data[1] = (byte) (intData >> 16);
                    data[2] = (byte) (intData >> 8);
                    data[3] = (byte) (intData);
                    //Write the data to the file
                    try {
                        fo.write(data);
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
                try {
                    fo.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        valueList.clear();
        valueListSize = 0;
        if(source.compareTo("Full") == 0) {
            sensorNames.clear();
        }
        timesWrittenToFile++;
        System.out.println(timesWrittenToFile);
        text.setText("Data collection runs completed: " + timesWrittenToFile);
        read = true;
        if ((timesWrittenToFile >= maxDataCollectionRuns) && maxDataCollectionRuns
            != -1)
            stopApp();
    }

    /*
     * Manages the copying from the output field
     */
    @Override
        public void onCreateContextMenu(ContextMenu menu, View v,
                ContextMenuInfo menuInfo) {
            menu.add(0, v.getId(), 0,
                    "Copy");
            ClipboardManager clipboard = (ClipboardManager) getSystemService(
                CLIPBOARD_SERVICE);
            clipboard.setText(text.getText());
        }
    @Override
        public boolean onOptionsItemSelected(MenuItem item)
        {
          //check selected menu item
          if(item.getItemId() == ID_MENU_EXIT)
          {
            //close the Activity
            this.finish();
            return true;
          }
          return false;
        }


}
```