Toward Sensor-Based Random Number Generation for Mobile and IoT Devices

Kyle Wallace, Kevin Moran, Ed Novak, Gang Zhou, Kun Sun

***Abstract*—The importance of Random Number Generators (RNG) to various computing applications is well understood. To ensure a quality level of output, high-entropy sources should be utilized as input. However, the algorithms used have not yet fully evolved to utilize newer technology. Even the Android Pseudo Random Number Generator (APRNG) merely builds atop the Linux RNG to produce random numbers. This work presents an exploratory study into methods of generating random numbers on sensor-equipped mobile and IoT devices. We first perform a data collection study across 37 Android devices to determine two things - how much random data is consumed by modern devices, and which sensors seem capable of providing sufficiently random data.**

We use the results of our analysis to create an experimental framework called SensoRNG, which serves as a prototype to test the efficacy of a sensor-based RNG. SensoRNG employs collection of data from on-board sensors and combines them via a lightweight mixing algorithm to produce random numbers. We evaluate SensoRNG with the National Institute of Standards and Technology (NIST) statistical testing suite and demonstrate that a sensor-based RNG can provide high quality random numbers with only little additional overhead.

***Index Terms*—Random Number Generation, Mobile Computing, Sensors**

I. Introduction

Random numbers and the generators thereof are an essential part of the mainstream computing landscape [1], [2]. The values produced by an RNG are utilized in a wide variety of applications, from OS-level functionality (stack pointer randomization), facilitating games and gaming content (AI decision making, lotteries, procedural generation), scientific computing (Monte Carlo, Markov models), and computer security (cryptographic key generation) [2]–[4].

While random number generation is a topic that has been well studied in the context of traditional computing environments, the rapidly growing mobile and Internet of Things (IoT) landscape has created a new space for research and exploration [5]. Mobile devices have proliferated and evolved into all-encompassing personal computers that not only perform familiar tasks, but also enable new functionality that standard computing environments are not equipped to address, such mobile payment and banking, or two-factor authentication. Meanwhile IoT-ready devices serve to extend the sensing capabilities of other devices, enabling previously 'dumb' technologies, such as the car or home, to become aware of their surroundings. This growing list of non-trivial use cases only adds to the demand for quality random numbers in a various contexts.

Many current RNG implementations either directly use - or are built on top of - the Linux PRNG (LPRNG), which draws its randomness from system level events and user input [6], [7]. However the LPRNG has difficulty extracting large amounts of entropy from these events, and instead relies on a large amount of mathematical mixing to produce random numbers [8]. To address this, there has been growing support for integrating hardware-based RNGs or alternative entropy sources in recent devices, such as with Intel RDRAND [9], [10]. However it is impossible for legacy devices to take advantage of newer hardware. Furthermore hardware is susceptible to problems such as bias, degradation, or backdoors, and is typically more difficult to fix should problems arise.

As a compromise between these two approaches, previous work has looked into extracting randomness from different sensors, such as the accelerometer or camera [11], [12]. However, these works are limited in their approach. Some are simply limited in the number of sensors they examine [11]–[13], limited in the scope of their analysis, or have analysis methods not suited for implementation in a mobile or IoT context. Others have not considered the impact of changing environmental contexts or hardware [11]–[14]. Furthermore, very few works consider the overhead of using sensors as an input source in terms of power use and CPU overhead [11], [15].

Based on the limitations of previous work, we chose the following research questions to address with our exploratory study.

RQ1 *Which sensors in modern mobile or IoT devices are capable of providing randomness, and how much?*

RQ2 *What is the demand for randomness in the context of a mobile system?*

RQ3 *How does hardware diversity impact the effectiveness of a sensor-based RNG?*

RQ4 *What kind of overhead does a sensor-based RNG impose on a mobile or IoT system?*

In summary, the major contributions of our work are as follows:

1) *We conduct a data collection study surveying 37 Android devices. Our analysis of the data reveals two things: which sensors are suitable sources of random noise*

Kyle Wallace, Kevin Moran, and Ed Novak are graduate students with the Department of Computer Science at the College of William and Mary. (*kmwall@cs.wm.edu, kpmoran@cs.wm.edu, ejnovak@cs.wm.edu*)

Gang Zhou is an associate professor with the Department of Computer Science at the College of William and Mary. (*gzhou@cs.wm.edu)*

Kun Sun is an assistant professor with the Department of Computer Science at the College of William and Mary (*ksun@cs.wm.edu*)

and the demand for random data in mobile devices. Specifically, we show that random data use tends to occur in short bursts, but never overwhelming to the RNG.

2) *We implement SensoRNG, a proof-of-concept RNG which draws randomness from hardware sensors. Our framework leverages opportunistic collection of data to efficiently gather the necessary sensor samples with reduced overhead. SensoRNG is implemented both as an Android system service, as well as an Android library for the sake of evaluation.*

3) *We provide an evaluation of SensoRNG on multiple aspects, demonstrating the viability of a sensor-based RNG as well as evaluating its overhead.*

4) *We discuss and provide insight into our findings, including the strengths and drawbacks of utilizing a sensor-based RNG.*

II. Background

A random number generator is effectively a black box that takes input and produces unpredictable numbers within some defined range. RNGs can be classified into two main categories - Pseudo-Random Number Generators (PRNGs) and True Random Number Generators (TRNGs). A PRNG is a complicated mathematical function that simulates randomness and is designed to be exceptionally difficult to reverse engineer based on output alone. The randomness of a PRNG stems from some random source, often referred to as a *seed*. A TRNG relies on an input source that is shown to exhibit random tendencies, such as radioactive decay or atmospheric noise, to produce values. Mathematically proving that a stream of bits produced by an RNG is *truly* random is effectively impossible. However it can be strongly suggested through rigorous statistical testing that a stream exhibits properties similar to what would be expected from a probability distribution [16].

1) Entropy: Entropy is a standard metric in information theory that measures the uncertainty of events in a probability space [17]. In the context of RNGs, we utilize entropy (in part) to describe how random a given stream of values is. To take an explicit measurement, we utilize the standard Shannon Entropy formula

$$H(P) = -\sum_{i=1}^{n} p_i * \log_2(p_i)$$

where p_i is the probability of a given event in P occurring. In the case of a random bit stream, the events in the probability space are all length k binary strings, and the probability of an individual event is equal to the number of instances that a particular string appears as a sub-sequence of the original bit stream. Shannon entropy is calculated against a uniform distribution and is reported in a unit of bits.

2) Applications: Random numbers have a wide range of application scenarios, from high-level user level applications to-low level system functions. High level applications fields such as scientific computing use random numbers when performing simulations. For example, an RNG could be used to initialize the parameters at the beginning of an experiment, or perform a sampling from potential items during. At the OS level, various constructs such as Address Space Layout Randomization (ASLR), stack canaries, establishing network connections, and much more.

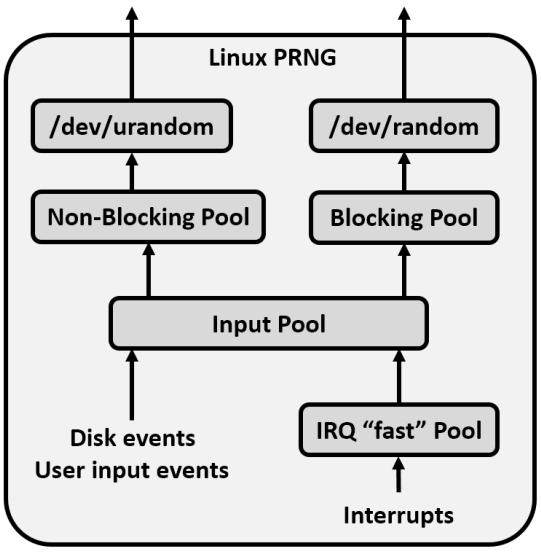


Fig. 1. The Linux PRNG framework. User input events correspond to keyboard and mouse input, or user touch events for mobile devices.

While the concept of applying random numbers is relatively straightforward, the consequences of a poor RNG varies from application to application. For something as simple as a game of chance, it can lead to simply poor user experience. In a scientific simulation, this can lead to lost time, or even false trends within the data. But for security algorithms, a poor RNG can result in vulnerability to attacks or data breaches.

3) Linux PRNG: Figure 1 details the architecture of the Linux PRNG (LPRNG). The LPRNG draws randomness from three main sources: user input (mouse and keyboard for desktops, touchscreen events for phones), interrupt request (IRQ) timings, and disk read/write timings. These events are collected by two pools, and then are fed into two output pools as needed. When the non-blocking pool `/dev/urandom` is read from, it will attempt to provide randomness from either the non-blocking pool or pull in fresh randomness from the input pool. If there is none available, it will use stale data from the non-blocking pool in order to produce randomness on demand.

At its core, the Android PRNG (APRNG) is an extension of the Linux PRNG, utilizing random data from `/dev/urandom` and hashing it to produce random values. The APRNG consists of two main parts: the `EntropyMixer`, and the `SecureRandom` front end. The purpose of the `EntropyMixer` is to preserve the current state of `/dev/urandom` on shutdown and restore it on boot. Additionally, it occasionally writes device-specific data to `/dev/urandom` such as the current time and the serial number. The other component, `SecureRandom`, acts as a front-end to the current PRNG algorithm `SHA1PRNG`, and is the current provider of cryptographically-secure random numbers for Android OS.

III. Related Work

We categorize related work as follows: exploratory studies into sensor randomness, methods of generating randomness in devices, and studies into the Android/Linux PRNG.

Sensor Randomness: The study carried out by Krkovjak *et al.* [11] investigates the microphone and camera in smart phones as promising sources of randomness. Similarly, Suciu *et al.* [12] study four sensors - the gyroscope, accelerometer, magnetometer, and GPS - to determine the level of randomness that each might provide. While Krkovjak et al. rely on Shannon entropy to quantify the non-deterministic nature of the sensors, we perform a deeper analysis to determine the significance of each bit per sensor sample. For the work by Suciu *et al.*, very little insight or information is provided about the utilized analysis methodology. The authors also only give a brief overview of how they combined incoming sensor streams. By comparison, we offer a detailed examination of a breadth of sensors examined in previous works. We also explicitly outline the architecture of our prototype, SensoRNG, and provide a detailed analysis of performance and power in comparison with the APRNG.

New Methods for Randomness Generation: Randomness generation in IoT devices has typically relied on the LPRNG. However several authors have proposed alternative methods for harvesting entropy or producing randomness. Kesley *et al.* proposed the Yarrow RNG as a general purpose solution, and is currently used in iOS and OSX [18]. In 2006, McEvoy *et al.* proposed the Fortuna PRNG as a cryptographically secure solution for generating random numbers. It has recently been adopted by FreeBSD [19]. Both of these algorithms could potentially be utilized in an IoT setting, but there has been no investigation into the potential of overhead.

More recently, Intel has begin adding support for hardware entropy gathering within the CPU with their RDRAND instruction [10]. Other work has has suggested that CPU jitter could serve as a suitable entropy source for generating random numbers [20], [21]. However, the former is limited to x86 processors while the latter has not received extensive testing on low-power devices.

With regards to sensors, Francillon *et al.* proposed a method for using received bit errors as a source of randomness in wireless sensor nodes [22]. Lo Re *et al.* proposed a method of using the physical measurements collected by large scale wireless sensor nodes as an input to a TRNG [23]. Our primarily concern in this work is with randomness extracted from commodity sensors available in mobile and IoT devices. We use these approaches as motivation for choosing which sensors to consider for analysis in our data collection study.

Studies on the Linux PRNG: The Android PRNG utilizes the Linux PRNG as part of its current implementation. There has been recent work done outlining the architecture of the LPRNG by Gutterman *et al.* [8] in 2006 and Lacharme *et al.* [7] in 2012. There are three major sources that Android uses to feed the random pool of the LPRNG - disk timings, interrupt timings and user touch events. However, in the study conducted by Ding *et al.* [14] it was noted that Android tends to rely heavily on disk events, especially directly after system boot. Furthermore, the amount of random bits that can be extracted from a single sample of one source is small, corresponding to 3 bits for disk events and 4 bits for interrupts [7]. Our study finds that a single sensor sample can provide much more.

TABLE I
Summary of the sensors chosen for study. GPS sample rate depends on movement, while camera sample rate depends on hardware.

Sensor	Length (bits)	Samples/second
Microphone	16 (x1)	44100
Accelerometer	32 (x3)	5
Magnetometer	32 (x3)	5
Gyroscope	32 (x3)	5
Radios	32 (x1)	2
GPS	64 (x2)	Variable
Camera	32 (x1)	Variable

Another important feature of the LPRNG is the entropy estimation counter associated with each pool. When data is added to a particular pool, the counter is incremented accordingly, and vice versa. These counters are kept for both the `random` and `urandom` pools. A recent analysis performed by Dodis *et al.* suggests that an attacker can take advantage of the manner in which these counters are implemented and potentially compromise the integrity of the output [24]. While our work does not explicitly investigate the security of the PRNG, we use works such as these as motivation for our exploratory study.

IV. Data Collection Study

This section outlines the details of our data collection study, in which we gather data traces from the entropy counter and various sensors. We target Android for ease of collection from a variety of sensors and devices, all of which run on top of the Linux kernel.

A. Study Overview

Modern Android devices come equipped with hardware sensors that are available for a variety of tasks. For example, many devices come with a microphone to enable the user to make calls and record audio, or an accelerometer to detect device orientation. With respect to a sensor-based RNG, we are interested in three sensor properties: the sample size (how many bits are needed to represent the sample data), the sensor resolution (the smallest change in value that a sensor can detect), and the sampling rate (how fast a sensor can report samples). Ideally, we want all of these attributes to be as large as possible.

Sensor Data: For our data collection study, we chose to include seven sensors commonly found in Android devices. Table I summarizes the sample size and rates for each sensor. The number in parentheses represents the number of axes the sensor reports on. These sensors were selected based on availability and the accessibility from an Android application. Documentation for interfacing with Android sensors can be found at the Android developer website [25].

Entropy Counter Data: The Linux PRNG tracks the amount of random data available for the system to use

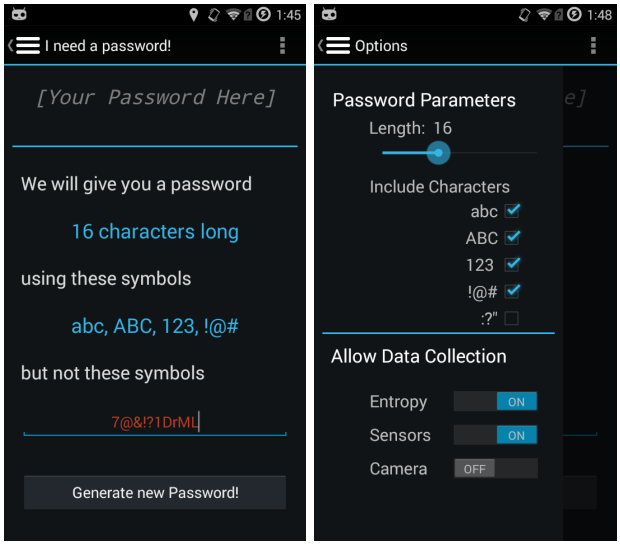


Fig. 2. Screenshots of the SensorPass app used for data collection.

when generating a value. This amount is stored in the file `/proc/sys/kernel/random/entropy_avail`, referred to as the entropy counter. The entropy counter is an estimate of the number of bits of randomness currently available in the input pool, and will increment and decrement accordingly when entropy is added or removed. The maximum amount of random data that can be stored at any time is 4096 bits. We choose to sample the entropy counter every 0.25 seconds.

B. SensorPass Application

To facilitate data collection, we implemented and distributed an Android application called *SensorPass* on the Google Play store, targeted at devices running at least Android 4.0.0. SensorPass consists of two major components - the front-end for the user to interact with and the back-end responsible for automating data collection. Figure 2 shows two screens of the user front-end.

The back-end to SensorPass is implemented as an Android Service, and consists of a number of auxiliary classes that collect data from each sensor. Collection is scheduled to execute every hour, determined by when the application is first launched. Data is collected from each sensor for three minutes, after which the service automatically stops collection and attempts to send data to our server. We only attempt to send over a Wi-Fi connection to avoid unnecessary use of a user's mobile data plan.

Due to the way Android implements the camera API, it is only possible to gather image data from the current active application screen. This is understandable from the standpoint of privacy, as malicious apps could take pictures or record video without alerting the user. Therefore we rely on asking users to manually collect camera data for us by interacting with a toggle in the options menu. When the user presses the toggle, we collect preview frames until exactly 1MB of data has accumulated, after which collection is automatically halted.

TABLE II
SUMMARY OF SENSOR DATA FROM SENSORPASS

Sensor	Total Data (Kb)	Num. Traces
Microphone	6,320,048	2288
Accelerometer	62,296	2313
Magnetometer	55,024	2306
Gyroscope	53,064	2182
Radios	48,356	2311
GPS	2,560	2315
Camera	144,036	69

Legal Notice: This user study was approved by the Institutional Review Board (IRB) at the College of William and Mary with PHSC protocol number *PHSC-2014-07-22-9695-gzhou*. Users were aware that data was being collected for research purposes, and all user data was kept anonymous.

Collection Statistics: Table II summarizes the data collected over the course of the study. In total we collected data from 37 devices running versions of Android ranging from 4.0.0 ("Ice-Cream Sandwich") to 4.4.4 ("Kit-Kat"). The total amount of data collected is 6.5GB. We note that a majority of the data collected comes from the microphone. This is because the sampling rate of the microphone is orders of magnitudes higher than that of the other sensors. We also note that the amount of data collected from the GPS is very low. This could be due to two factors. First, users may not have turned on their GPS during collection, resulting in no values being reported. We also only collect data when the user's location has changed more than one meter, as interval polling resulted in too many duplicate values. Under this strategy, a user not in motion would only report one or two values.

C. Analysis Methodology and Tools

Sensor Data: The main objective in analyzing the sensor data is to extract sufficient randomness from the samples for further use. As illustrated in Figure 4, our approach takes a bit-wise investigation of each sensor by treating successive samples in each bit position as individual data streams. We chose this analysis method for two reasons. First, directly examining the raw bits requires the least amount of computation, as opposed to performing more in-depth data analysis. This also eases the burden of processing when extracting randomness in the framework. Secondly, it allows us to use a general framework for sensor analysis, rather than requiring new methods for individual sensors. This allows for additional sensors not covered in this work to be easily examined in future work.

For analyzing the randomness of a given stream, we utilize the NIST *Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*.[1] The NIST suite is freely available to the public, open source, and provides a straightforward framework for determining whether or not a given stream of bits or numbers appears statistically random. We refer to a RNG under test as an *input source*, while a string of random data produced by the generator as an *input stream*.

[1]http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html (as of Nov. 2015)

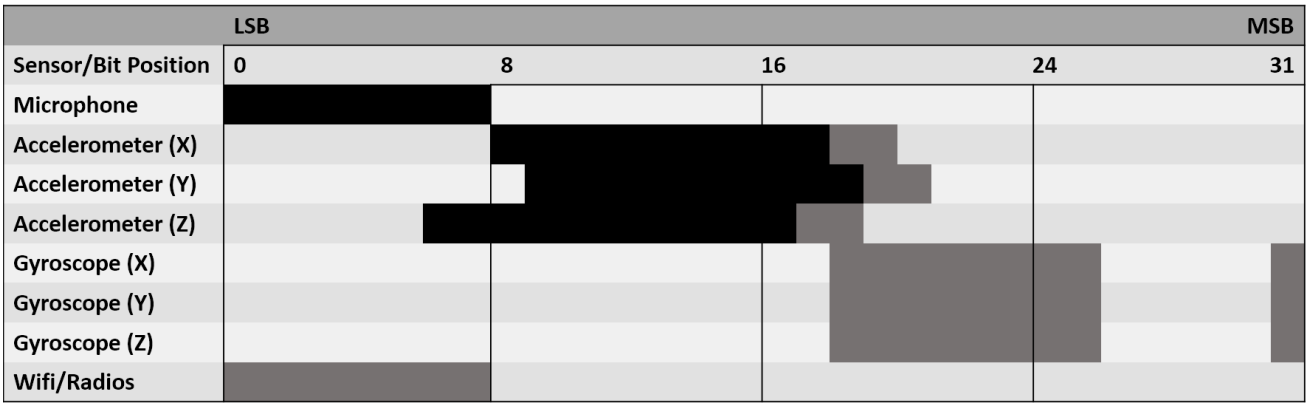


Fig. 3. Heatmap of which bits from sensor samples show sufficient randomness. A black square indicates the bit is 'good', a gray square indicates a bit is 'fair', and an uncolored square indicates the bit is 'bad'. We have excluded the Magnetometer, GPS, and Camera rows as they provided 0 good bits.

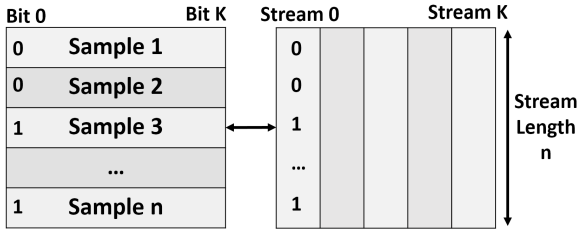


Fig. 4. Diagram of the bitwise method used for analysis. The left block represents successive samples from a sensor (horizontal), while the right block represents the k streams we form for analysis with the NIST suite (vertical).

For a given input source, the full NIST Suite performs a battery of 15 statistical tests, each designed to evaluate a certain property of a single input stream against how that property would manifest in a uniform random stream. For each single run of a test, a p-value is returned which indicates whether or not the stream passes that particular test. A p-value greater than 0.05 is considered passing, indicating that the stream is *not significantly distinguishable* from random. Running a test on multiple streams from the same source produces a collection of p-values which can be characterized by a distribution, on which the final reported p-value is computed. For a *source* to be considered truly random, this distribution of p-values should tend toward completely uniform, *implying that some individual runs of a test will fail*.

For the purpose of our analysis, we pick a subset of 7 tests from the full NIST suite - the frequency test, frequency test within a block, runs test, longest run of ones within a block, DFT test, binary matrix rank test, and approximate entropy test. We specifically pick these tests to act as a simple sanity check for 'good' and 'bad' bits. Each test addresses a different quality of randomness - for example, the rank test make sure there is no periodicity in the data. Complete descriptions of each test and how to interpret the results can be found in the NIST suite documentation [16].

Entropy Data: Our main goal in analyzing the entropy counter traces is to assess the current demand for random data by the APRNG. We want to observe any patterns in random data use to help guide the design for a sensor-based RNG. The data collected takes the form of integer samples over time. Therefore we treat each collected entropy trace as a time series for analysis and compute general statistics such as median, mean, and max. Furthermore, we estimate the amount of random data used over the entire trace by summing up all the instances of a drop.

V. Data Analysis Results

This section presents the analysis and results of the data gathered in our collection study. We first begin with analysis of the sensors, and then cover the analysis of random data use.

A. Sensor Data

This section presents the results from analysis of the collected sensor data. We use a three tier classification to determine which bits are the best candidates for use in SensoRNG. For a given bit to be 'good', it must pass at least 3 of the NIST tests at least 75% of the time. For a bit to be considered 'fair', it must pass 1-2 tests at least 75% of the time, or at least 3 tests at least 50% of the time. A 'bad' bit is any bit that is not good or fair. In the implementation of SensoRNG, the utilization of "good" bits is preferred over the utilization of "fair" bits. These numbers were chosen empirically, with the intuition that while individual bit streams may not provide enough entropy on their own, mixing together several streams will mask or eliminate any individual deficiencies. (I.E. It should only take roughly 2-4 'good' bits or 4-8 'fair' bits to produce one usable bit of entropy)

Figure 3 illustrates the results of our analysis in a heat map. Note that some sensors under test have been excluded due to poor results. Some of the sensors that were cited as good candidates for randomness in previous work (such as the camera) do not perform as well under our analysis [11], [13], [26]. This is likely due to the difference in techniques, as examining bits individually is not tailored to any particular data type. While this does not mean the particular sensor is unusable for the production of random numbers, it does indicate that the computational effort necessary to extract randomness will likely be greater.

Summary of Findings: Overall, the data suggests that the microphone is the best candidate for extracting usable amounts of random data, producing 8 good bits per sample at a very high rate. Following this is the accelerometer at 31 good bits

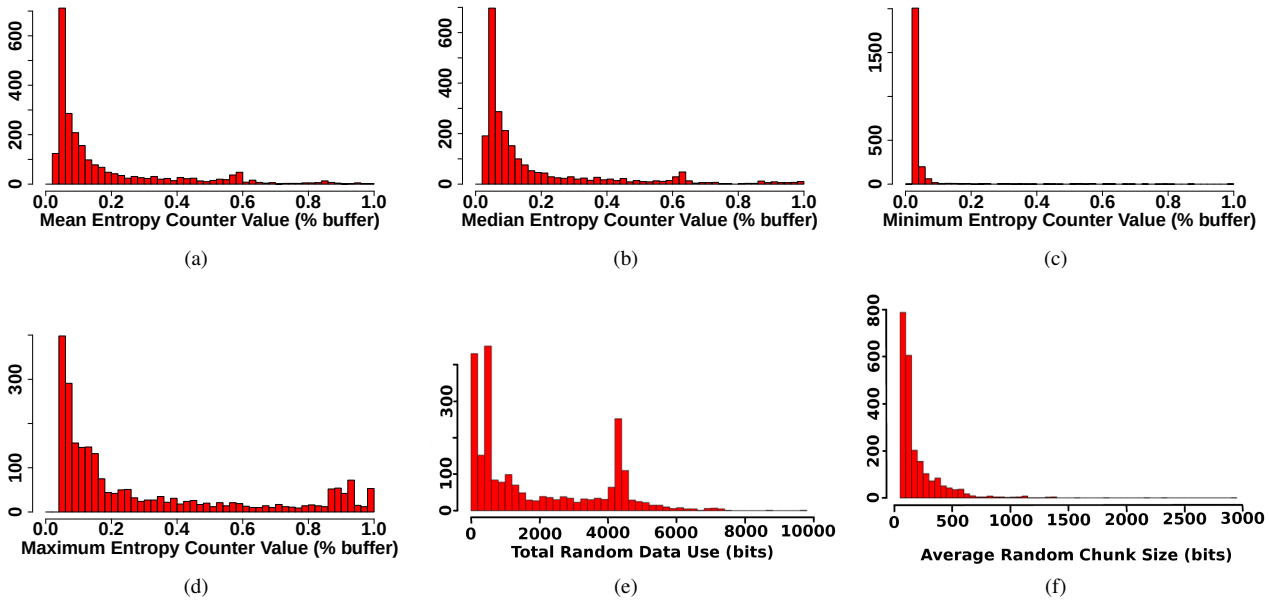


Fig. 5. Distributions of entropy trace statistics. The Y axis is measured in number of traces

TABLE III
QUANTILES OF MEASURED STATISTICS ACROSS ALL TRACES. VALUES LISTED ARE IN BITS.

Quantile	0%	25%	50%	75%	100%
Mean	159	203	349	763	4096
Median	157	200	330	686	4096
Minimum	7	128	131	138	4096
Maximum	174	308	588	1690	4096
S. Deviation	0	39	119	384	1434

TABLE IV
STATISTICS OF TOTAL RANDOM DATA USE ACROSS ALL TRACES. VALUES LISTED ARE IN BITS

Statistic	Mean	Min	Median	Max	S. Dev
Total	1873	0	961	9644	1850
Avg. Mag.	195	50	117	2922	207
Quantile	0%	25%	50%	75%	100%
Total	0	415	961	3891	9644
Avg. Mag	50	73.1	117	232	2922

per sample, but at a lower rate. The gyroscope follows the accelerometer by providing 27 fair bits per sample, however a gyroscope is not guaranteed to be present in every device. The radios follow, providing only 16 fair bits per sample. We find that the magnetometer and GPS are not considerable sources of randomness, though there is further room for investigation into the GPS due to a small sample size. Similarly, we are unable to extract any usable bits from the camera, likely due to the analysis methodology.

B. Entropy Counter

This section presents analysis of the entropy counter traces. Recall that the data collected for this part of the study consists of an integer-valued time series with a rate of 4 times per second. Figures 5a-5d plot histograms detailing the distribution of values for four metrics across all traces - mean, median, minimum, maximum. We find that each statistic roughly follows a negative exponential distribution, implying that either a majority of devices are actively using random data during the sampling period, or that the pool of random data tends to only refill gradually. Table III further summarizes the quartiles of each statistic.

Total Entropy Use: Figure 5e illustrates the distribution of total random data use across all traces, while Table IV summarizes basic statistics about the distribution. Over the course of a 3 minute trace, we calculate approximately 10 bits of randomness being used per second on average, and less than 5.3 bits of randomness being used per second in 50% of scenarios. However the standard deviation is rather large, indicating that there may be rare periods of heavy demand. The observed maximum rate of random data use is approximately 53.5 bits per second. This rate is easily sustainable with only a few sensors being turned on. We note that there is a cluster of traces all using around 4096 bits, which is the total size of the buffer for the APRNG. However, we were unable to determine the cause of this phenomenon.

Magnitude of Use: Figure 5f illustrates the average magnitude of random data use. To calculate this, we summed up all instances where the entropy counter dropped and divided that value by the number of instances of the counter dropping across the trace. We merged together contiguous drops to count as one instance. This represents the average size of a request for random bits. Table IV summarizes the findings. We note that in a large majority of cases, the magnitude of a request is less than that of 8 integers (256 bits), which indicates that random data is typically only needed in short bursts. Only in very rare cases are larger requests made, but no request is big enough to drain the buffer completely.

Summary of Findings: In our investigation, we find a stratification of random data use patterns. On one hand, half of the traces report very low values, indicating that the device is idle or experiencing light use. On the other hand, random data use falls into two main categories - constant, light use or heavy, incidental use. While roughly the same amount is used at the end of the sampling period, the shape of these plots

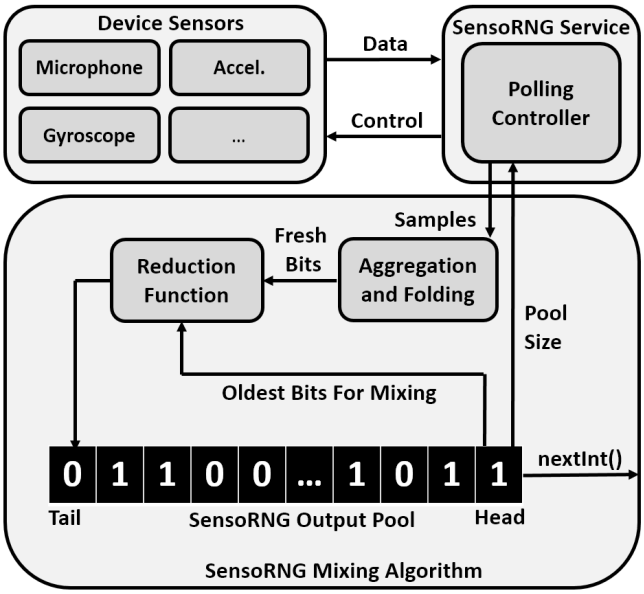


Fig. 6. The SensoRNG Framework. Input is received from sensors via the polling controller and then queued for processing. Processed samples are merged with values already present in the buffer and then sent through a reduction function to further mix together temporally separate bits.

are vastly different. Overall, we find that the need for random numbers is always present and experiences occasional spikes.

VI. SensoRNG

We now present the framework for SensoRNG, our proof-of-concept sensor-based RNG. Figure 6 presents the architecture of the algorithm. Using the assumption that the data from sensors provides a minimum guarantee of randomness, our design of SensoRNG is kept intentionally simple. There are three main components - the controller, the aggregation and folding function, and the reduction function, which serve the roles of collecting samples, processing and combining samples, and mixing entropy into the buffer respectively. We utilize two layers of mixing via the aggregation and reduction in order to fold together randomness that is both temporally local and temporally distant.

We implement two versions of SensoRNG for the purposes of evaluation. The first version is a system service embedded in Android OS. Here, we instrument the sensors directly to facilitate opportunistic collection of sensor streams without introducing additional polling overhead. Opportunistic collection has been utilized in other works to minimize the energy overhead of collection [27]. The second version is an Android application library. Instead of opportunistic collection, we instead utilize reactionary collection, manually polling only when the internal buffer drops beneath a threshold of 25%. We instantiate two versions to evaluate 1) the quality of the output produced and the overhead in terms of power; and 2) The ease of adapting our framework to existing applications respectively.

Polling Controller: The controller is the component that acts as the middleman between the hardware sensors and the SensoRNG mixing algorithm. The duties of the controller are threefold: First, it serializes incoming sensor samples and processes them, stripping them down to the most desired bits as determined in section 5. Second, it monitors the amount of data available in the random buffer, ensuring that it stays above the minimum desired capacity. Should passive collection of sensor data fail to meet the needs of the system, the controller can briefly turn on any sensor in order to help refill the buffer to an acceptable level. We discuss specific implementation parameters in the evaluation section.

Aggregation and Folding: This routine is called by the controller in order to process individual sensor samples. In this function, non-random bits are stripped away and the remaining are compressed into a smaller stream of information based on the results of our sensor analysis. Specifically, we split each incoming sample into two sets, G and B, where G consists of all good bits, and B consists of all 'bad' bits. Instead of directly using G, we take the parity of all bits in B and reverse the order of the bits in G if the result is 1. This serves simply as an occasional additional step in the mixing function

The next step, the *aggregation* step, we store the results of the previous step ($E = G_1G_2 \dots G_k$) in a processing queue. Once enough samples have been collected, we create a bitstring T of fixed length l for the *folding* step. The algorithm then pops the top element E from the processing queue and 'stripes' it across T. Namely, let T have a position pointer p. Then for each bit i in E, we perform the following operations

$$T[(p+i) \bmod l] = T[(p+i) \bmod l] \oplus E[i]$$

Where $\oplus$ is bitwise xor. This process is repeated for a number of samples $E_1, E_2 \dots E_n$. Once this process is complete, T is sent to the reduction function.

Reduction Function: The reduction function takes input from both the internal buffer and the folding function in order to further mix together bits that are not temporally local. We take inspiration in our design from asymmetric cryptography algorithms which utilize a substitution table, or 's-box', to mix in key bits [28]. We aim to make the reduction function difficult to reverse to prevent reconstruction of input data, ensuring backwards unpredictability. This is realized by using a 'many-to-one' mapping, where multiple inputs map to a single output.

The reduction function operates as follows: Inputs to the function are three n bit chunks, T, H_1 and H_2 corresponding to freshly processed data, and the first two n bit chunks from the head of the buffer. We first calculate $I = T \oplus H_1$. I serves as input to a substitution table in order to get output S. The length of S in bits can vary based on the parameters used to generate the table. We then concatenate together $H_2, S, \neg H_1$ and append the result to the end of the buffer, shuffling the order and parity of bits that were already in the buffer.

The substitution table is generated using the following procedure. There are three parameters - input length n, minimum output length m, and output length range r. First, we generate a random permutation of the integer values in $[0, 2^n)$. We then form a sorted list of bit strings between length m and length $m + r - 1$. Starting from a random point in the permutation, we step through both the permutation of values and the list of bit strings, creating pairs and storing them in a hash table.

TABLE V
EXAMPLE SUBSTITUTION TABLE `R(x)` IN THE REDUCTION FUNCTION WITH PARAMETERS $(n, m, r) = (4, 1, 2)$. FOR BREVITY, INPUT IS LISTED IN HEXADECIMAL, WHILE OUTPUT IS LISTED AS A BINARY STRING.

x	1	3	5	7	9	B	D	F
R(x)	0	1	00	01	10	11	0	1
x	0	2	4	6	8	A	C	E
R(x)	00	01	10	11	0	1	00	01

An example substitution function `R(x)` is in Table V. The table used in the SensoRNG algorithm is generated randomly with the first few incoming bits. Note that by this design, multiple input values can map to the same output value. Similarly, by varying the output length, it is difficult to tell what segments in the output map back to input segments.

Theoretical Complexity: The SensoRNG algorithm is designed to be computationally lightweight with a theoretical complexity of $O(n)$, where n is the number of bits in a given input. Consider a single input of length n. Determining the good and bad bits of the input is done via a bitmask and shift, which results in two operations per bit, or at worst $2n$ operations. A reversing of the good bits due to the parity of the bad bits may result in another n operations. The aggregation and folding function performs an additional n bitwise xor operations to fold together successive samples. In the reduction function, there is one bitwise xor of two n bit strings, one negation of an n bit string, and one substitution in a hash table for $O(1)$. In total, this brings the theoretical complexity to $6n + O(1)$, or $O(n)$.

VII. SENSORNG EVALUATION

In this section, we present our experimental evaluation of SensoRNG in comparison with the current Android OS implementation of `SecureRandom`.

A. Experimental Setup

We pick two main targets to evaluate SensoRNG: quality of random numbers provided and the power efficiency of each implementation.

Quality: To evaluate the quality of the random numbers returned by SensoRNG we once again employ the NIST suite, utilizing a larger subset of tests in order to rigorously evaluate produced bit streams. In addition to the seven tests used for sensor analysis in section 5, we also include the cumulative sum, serial, and linear complexity tests [16]. We exclude the non-overlapping template test, the overlapping template test, Maurer's "Universal Statistic" test, and the random excursions test due to the large number of potential parameters.

Power Efficiency: To evaluate the power consumption of each RNG, we perform measurements under two scenarios by simulating the statistical average and maximum random data usage found during our analysis in section 6. This is done by periodically making a call to `getRandomBytes()` at a the appropriate rates - 10 bits per second and 55 bits per second respectively.

To take power measurements, we utilize the Trepn power monitor for Qualcomm Snapdragon processors [29]. For each sensor we profiled a small test-harness application that independently polled the microphone, accelerometer, and gyroscope at the frequencies used for SensoRNG. We also used the harness to profile each device while generating random numbers. When profiling, we used the "Profile App" feature of the Trepn power monitor with all overlays turned off. We collected only the Power Measurement data point, with a sampling rate of 100 ms. The Trepn Profiler has been utilized in related research for accurately taking power measurements [30], [31], and it has the ability to isolate and profile on a per application basis.

B. SensoRNG Implementation

For our prototype implementation of SensoRNG, we utilize the three most promising sensors discussed in this paper - the microphone, gyroscope, and accelerometer. Based on our analysis, these provide the most random data per sample and have acceptable rates to cover established needs. We also note that the accelerometer is constantly being polled at a low rate by Android OS, likely to detect screen rotation. This was discovered during instrumentation of Android OS.

To implement the entropy controller, we utilize a set of simple thresholds, similar to how the Linux PRNG operates. The length of the internal buffer for SensoRNG is set to 4096 bits long, the same as the Linux PRNG. When the internal buffer falls below 25% capacity, we manually begin polling the gyroscope and accelerometer to compensate. If the internal buffer falls below 128 bits, we begin manually polling the microphone. Should both of these methods fail to refresh the buffer, we choose to block the call for data in order to provide sufficient randomness. Once the pool has refilled beyond 95% capacity, we switch off any manual polling to save on power. For the substitution table in the reduction function, we choose an input length of 8 bits, and an output length ranging from 2-4 bits.

Devices: All tests are performed on a Nexus 4 and Nexus 5 running Android OS 5.0.1 "Lollipop". For the `SecureRandom` tests, we utilize the factory images available from Google.[2] For the SensoRNG tests, we utilize a modified version of the Android 5.0.1 source compiled for each device where `SecureRandom` is instrumented to utilize SensoRNG.

To generate the streams for testing, we wrote a small testbed application that periodically makes calls to the `getRandomBytes()` method for both `SecureRandom` and SensoRNG. All experiments are performed with wireless turned off and the screen at minimum brightness to minimize energy noise. Similarly, as the wireless radios were not used in SensoRNG, the SIM card was removed. Collection of random numbers takes place during two scenarios: an 'idle' scenario where the device is sitting in a quiet office environment, and a 'typical' scenario where the device is in a pocket and experiences light use during the day.

C. Evaluation Results

We now present the results of our evaluation of SensoRNG in comparison to the APRNG's `SecureRandom`.

[2]https://developers.google.com/android/nexus/images

TABLE VI
COMPARISON OF REPORTED P-VALUES FOR SENSORNG (SRNG) AND SECURERANDOM NIST SUITE RESULTS. EACH TEST CONSISTS OF 200 RUNS OF 40,000 BITS EACH. $\alpha = 0.01$ IS SIGNIFICANT. (F) AND (R) STAND FOR THE FORWARD AND REVERSE VERSIONS OF A TEST RESPECTIVELY. VALUES OF $p < 0.01$ ARE ITALICIZED AND MARKED BY ASTERISKS.

	Nexus 4			Nexus 5		
Test Name	SRNG (idle)	SRNG (typical)	S. Random	SRNG (idle)	SRNG (typical)	S. Random
Frequency	0.3881	0.5955	0.9357	0.9240	0.8255	0.7298
Block Frequency	0.0363	0.1718	0.5042	0.3838	0.7981	0.6267
Cum. Sum (f)	0.5442	0.1969	0.9470	0.7981	0.6786	0.2429
Cum. Sum (r)	0.2461	0.6838	0.4846	0.6890	0.4654	0.7887
Runs	*0.0048*	0.0303	0.5442	*0.0043*	0.0351	0.2968
Longest Run	0.9868	0.9681	0.8074	0.5749	0.3627	0.8074
Rank	0.0302	0.0117	*0.0005*	0.1188	0.3041	0.3669
FFT	0.7548	0.8676	0.2248	0.0205	0.0965	0.3586
Approx. Entropy	0.2248	0.5697	0.4372	0.0104	0.2133	*0.0007*
Serial (f)	0.9512	0.2622	0.8741	0.0689	0.2077	0.7597
Serial (r)	0.9178	0.4465	0.9463	0.6993	0.9733	0.8165
Linear Complexity	0.7695	0.3504	0.2248	0.7791	0.3753	0.2429

1) Quality: Table VI summarizes the results of the NIST suite for both SensoRNG and `SecureRandom`. The reported p-value is calculated based on the distribution of the results of all runs of a particular test. More information on the meaning of this value is provided in the NIST suite documentation [16].

Overall, we find that SensoRNG performs favorably against `SecureRandom`. Both implementations pass all but one test, with a typical scenario passing all tests for SensoRNG. In terms of individual tests we find the results to be split evenly, with SensoRNG reporting higher p-values in some instances and `SecureRandom` reporting higher values in others. We note that a higher p-value in terms of the NIST suite should be taken simply as a stronger statistical suggestion of randomness, not a binary comparison of 'better' versus 'worse'.

For some tests, SensoRNG has weaker p-values - particularly the *runs* test and *rank* test. This is likely a side-effect of the mixing function. The runs test checks to see how quickly a given stream oscillates between 0 and 1. Because one of the mixing function components is a substitution table, it is likely that large strings of 0's or 1's are being broken up, increasing the overall 'oscillation' of the bits in the output. This would also impact the reported values of the *approximate entropy* test and the *rank* test, which both look for large and small blocks of similar bits.

2) Power: Table VII briefly summarizes the power draw for polling each sensor on each test device. The numbers were computed as follows: For each sensor we perform no sampling for three minutes to get a baseline measurement. We then turn on the sensor for three minutes and sample at the default rate used in our data collection study, afterward subtracting out the baseline measurement to isolate the sensor power use. All values are in `mW`.

Across both test devices the accelerometer utilizes the least power of the three chosen sensors, followed by the gyroscope and then the microphone. For the Nexus 5, we find that turning on all sensors uses additional 51.5mW, for a total of 12.9%. For the Nexus 4, all sensors together only use an additional 70.1mW, or about 13.1% in our testing scenario. Despite the microphone using the most power, it also provides the highest sampling rate of the three sensors. This indicates that even though the microphone is more expensive in terms of power, it has a better power ratio for production of randomness.

TABLE VII
POWER VALUES FOR SAMPLING SENSORS AT THE DEFAULT RATE, PER TEST DEVICE. BASE+ IS A BASELINE MEASUREMENT WITH ALL SENSORS ACTIVE. ALL VALUES ARE REPORTED IN `mW`.

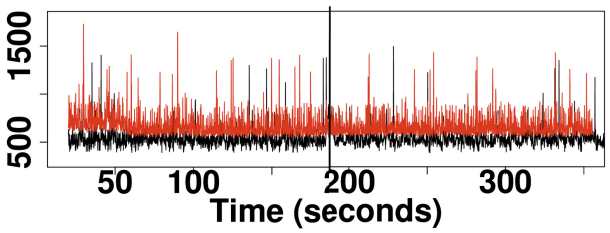
Device	Mic.	Accel.	Gyro.	Base	Base+
Nexus 4	32.8	10.3	27.0	534.6	606.7
Nexus 5	22.0	10.8	18.7	399.1	452.9

Figures 7a and 7b show the power traces of the test devices while they produce random numbers under two scenarios: average load (10 bits/second) and max load (55 bits/second). SensoRNG at the OS level employs opportunistic collection of sensor data whenever possible. This means that even though extra power is being drawn due to the sensors being on, SensoRNG is not responsible for the overhead of polling. To isolate the computational overhead, we took a measurement - indicated as 'Base+' in Table VII - that examines baseline power consumption with all sensors active. Against this adjusted baseline, we see that SensoRNG only uses an additional 10mW in the Nexus 5 for the average case, and 28mW extra for the Nexus 4, resulting in only a 2% and 4% increase respectively.

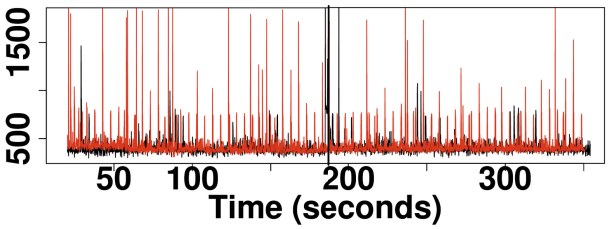
We also consider the worst-case for SensoRNG by considering it responsible for all additional power overhead. For the average load scenario, we find that the Nexus 5 uses an additional 31mW on average over `SecureRandom`, and the Nexus 4 uses an additional 82mW on average. This translates to a 7% increase in power consumption for the Nexus 5, and a 15% increase for the Nexus 4. For the maximum rate scenario we find that the power consumption increases, with the Nexus 5 using an additional 35mW on average and the Nexus 4 using an additional 94mW when compared to the baseline. This translates to a 8% increase in power for the Nexus 5 and a 17% increase in power for the Nexus 4. While this is a notable increase for the worst case scenario, devices should never be in this use state except for rare circumstances.

VIII. APPLICABILITY STUDY

To demonstrate the ability of the SensoRNG system to immediately impact real world Android applications, we implemented the framework in an Android Library called SensoRNGLib and modified 5 free and open source (FOSS) applications from the F-Droid marketplace [32], a popular and



(a) Nexus 4



(b) Nexus 5

Fig. 7. Power traces taken while producing random numbers. Black represents `SecureRandom` while red represents SensoRNG. The left half of each plot is the average (10 bits/sec) scenario, while the right half is the max (55 bits/sec) scenario.

well-maintained repository for FOSS Android apps. Our study targets two metrics to evaluate the applicability of SensoRNG to existing apps: 1) effort involved in adopting SensoRNGLib; and 2) the computational overhead of SensoRNGLib method calls.

A. SensoRNGLib Implementation

One important missing feature from SensoRNGLib is opportunistic collection of sensor data, which requires hooks into the sensor data streams. Instead, we utilize reactionary collection, where every time random data is requested we check the status of the random pool. If the request would drain the pool below a certain threshold, we activate all sensors for 3 seconds and then turn them off. We empirically determined 3 seconds to be sufficient to both refill the buffer and provide enough mixing to facilitate quality randomness. While true opportunistic collection cannot be performed, the API does provide a method for developers to pass sensor data into the library if their application already uses said sensors. These two features allow SensoRNGLib to operate in a similar fashion to the LPRNG, which uses simple thresholds and allows for processes to write to `/dev/(u)random`.

B. Developer Effort

We extracted 5 applications from the F-Droid marketplace in order to evaluate the programming effort required to adapt SensoRNG to real-world apps. When choosing these applications we aimed to fulfill several criteria including: 1) Apps that are popular or well-known (based on number of downloads or developer activity), 2) Apps of varying size and complexity (in order to offer a broad discussion of the programming effort required for different size apps), 3) Apps that contain at least one call to the system-level implementation of the

TABLE VIII
DEVELOPER METRICS FOR IMPLEMENTING SENSORNGLIB. TIME (IN MINUTES) IS MEASURED FROM THE START OF COMPILING THE ORIGINAL SOURCE SUCCESSFULLY TO COMPILING THE INSTRUMENTED VERSION SUCCESSFULLY.

Metric	RMP	k9	KeePass	Addi	aagtl
LoC Changed	5	8	21	8	5
Time (mins)	15	20	30	30	15

TABLE IX
AVERAGE NORMALIZED CPU USAGE FOR BOTH THE ORIGINAL AND SENSORNGLIB IMPLEMENTATIONS OF KEEPASSDROID AND RANDOMMUSIC PLAYER.

	KeepassDroid	RandomMusicPlayer
Original	24.07%	25.74%
SensoRNG	23.65%	24.92%

Random Number generator (e.g. calls to `SecureRandom`). Thus, as our subject applications we used: k9 mail [33], keepassdroid [34], RandomMusicPlayer [35], Addi [36], and aagtl [37]. For each of these applications we replaced the calls to the standard Android/Linux RNG with calls to the appropriate methods in the SensoRNGLib. To evaluate the programming effort required to adapt each application, we recorded the total number of lines of code changed and the time required to modify each app. Our experience indicates that modification of applications to utilize SensoRNGLib is very intuitive, requiring little effort on behalf of the developer even in complicated applications.

C. Computational Overhead

In order to evaluate the computational overhead of the SensoRNG implementation of each app to the original implementation, we profiled each application with the Android activity manager profiler (AMP) [38] in order to collect method traces for general uses of each application. We selected two applications (RandomMusicPlayer, and Keepassdroid), for which we could reliably (e.g. deterministically) construct GUI-based execution scenarios that trigger calls to the RNG. We then recorded the low-level GUI-event scenarios on a Google Nexus 5 smartphone using the `getevent` Android shell command [39] for each application alongside method traces to be sure that the recorded scenarios triggered the method calls related to the RNG. Next, we translated these low-level event traces into high level executable scripts in the form of `adb` commands (e.g. `adb shell input tap 507 565`) using a methodology inspired by RERAN [40]. After the translation, we replayed these event sequences for both versions (e.g. SensoRNG and original) of each app on the Nexus 5 device while collecting normalized cpu-usage information using the Trepn profiler [29]. When conducting these tests the phone's network connections were disabled and only the Trepn profiler and target application were running, with the Trpen profiler only targeting the specific app-under-test. This methodology should produce reliable results that isolate the performance recordings of the application in question. The results show no significant deviation in cpu-usage between the two implementations, suggesting that the SensoRNG implementation of these apps does not impose additional computational overhead.

TABLE X
COMPARISON TABLE FOR DIFFERENT SENSORNG IMPLEMENTATIONS.

OS Implementation	App Library
* System service, will always be available	* Service tied with the app process
Opportunistic collection of sensor data	* Selective, manual polling of sensors
Heavy load impacts total system performance	* One hungry app taxes its own buffer
Centralized buffer for all processes	* Individual buffers for each app
One buffer size for all processes	* Customizable, per-app buffer sizes
Requires OS modification, harder to adopt	* Easy to include in any app
One algorithm for all processes	* Can customize algorithm per app
Available to system processes	* Not available to system processes
System can securely store buffer on shutdown	* Apps must ensure secure buffer storage, extra effort

IX. DISCUSSION & FUTURE WORK

Limitations: One feature missing in sensor-based RNGs is the ability to reliably generate the same sequence of random numbers on demand. This capability is central to debugging and verification, as these activities require reproducible behavior, and a PRNG can simply utilize a test seed to easily reproduce a sequence of random values. To implement such functionality, the user would have to exactly recreate all sensor inputs in the same order - a feat that is physically improbable. A workaround is possible by allowing a "debug" mode that accepts input from a single source, such as a file. However this remains an inelegant solution.

One current limitation of SensoRNG is that our analysis of samples is done on a global scale across multiple devices. However, it may be the case that what works well for one device configuration is not the ideal case for another. For example, older devices may have a lower sensor resolution and provide fewer usable bits per sample. In the future, it would be worth designing methods to investigate devices on an individual basis, creating a 'device profile' that can characterize randomness from each sensor.

While we show it is possible to passively harvest sufficient entropy from sensors on mobile devices, smaller IoT devices may struggle to collect enough randomness to meet their own needs. This is entirely dependent on what sensors the device comes equipped with. Furthermore, the power cost of processing sensor samples may be too high for low-end devices, or devices with batteries, to tolerate. Because of this, future testing will target low-end devices to see if entropy needs can still be met, and if not, whether potential hybrid options can take advantage of the sensor as an entropy source while lessening the impact on battery.

Implementation Considerations: For our work, we implemented SensoRNG at two locations - in the OS as a system service, and in the application layer as an Android library. Table X illustrates a number of trade-offs we noticed during implementation and evaluation. We summarize these points under three main categories.

Performance: With regards to performance and overhead, we find that implementation at the OS level is more efficient. This is because there is only one buffer to track and one processing queue for samples. At the library level, each application gets an individual buffer to store random bits in. Similarly, each app is responsible for processing sensor data to extract randomness, rather than just the system. Consequentially, the power overhead can be slightly higher as the app library cannot rely on opportunistic collection unless the app itself uses the desired sensors. However, one app taxing the RNG at the OS level may impact performance system wide, whereas one app taxing its own RNG will not.

Flexibility: With regards to flexibility, we find that the app library is much more flexible for the needs of an app developer. Instrumenting a sensor-based RNG at the OS level requires modifying and recompiling Android OS, which is not possible for every device. However, an Android app library has documented support for inclusion into any app, making the bar for adoption much lower. Similarly, as we made the library open source, it is possible for anyone to modify the algorithm or parameters to their needs, whereas it would be much more difficult to modify at the OS level.

Feature Availability: With regards to feature availability, the OS implementation is slightly more robust. An RNG at the OS level can be available to all processes, while an RNG in an app library is only available to the processes that want to implement it. Similarly at the OS level, the buffer can be easily stored between boots, while it is up to the developer to choose whether or not to do so at the library level.

X. CONCLUSIONS

This paper presents an exploratory study into the viability of a sensor-based RNG for mobile and IoT devices. Our findings on the state of random data use in the Android PRNG show that, in the average scenario, devices operate under conditions of light, but constant use. Furthermore, we show which sensors on modern hardware are capable of meeting the demand for random data. To evaluate these claims we present a prototype framework SensoRNG, which exploits the noise in sensor data for the purposes of generating random numbers. Our evaluation on several points compares favorably against the current Android PRNG, with only a small computational overhead, strongly suggesting the viability of a fully optimized solution.

ACKNOWLEDGMENT

The authors would like to thank the members of the LENS Lab research group for their support and feedback throughout the lifetime of this paper.

This work is supported in part by the U.S. National Science Foundation under grant 1253506 (CAREER).

REFERENCES

[1] Wolfram, "Random number generation. http://reference.wolfram.com/language/tutorial/RandomNumberGeneration.html."

[2] "Random.org. http://www.random.org."

[3] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[4] W. Stallings, *Cryptography and Network Security: Principles and Practice*, sixth edition ed. Prentice Hall, 2013.

[5] A. Greenfield, *Everyware: The dawning age of ubiquitous computing*. New Riders, 2010.

[6] T. Vuillemin, F. Goichon, C. Lauradoux, and G. Salagnac, "Entropy Transfers in the Linux Random Number Generator," Grenoble Research Center, 655 Avenue de l'Europe Montbonnot, Tech. Rep. 1, September 2012.

[7] P. Lacharme, A. Röck, V. Strubel, and M. Videau, "The Linux Pseudorandom Number Generator Revisited," *International Association for Cryptologic Research*, pp. 1–23, 2012.

[8] Z. Gutterman and B. Pinkas, "Analysis of the Linux Random Number Generator," *International Association for Cryptologic Research*, pp. 1–18, March 2006.

[9] Apple, "iOS Security," Apple Inc., Tech. Rep., February 2014.

[10] "Intel rdrand https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide."

[11] J. Krhovjak, P. Švenda, and V. Matyáš, "The Sources of Randomness in Mobile Devices," in *Proceedings of the 12th Nordic Workshop on Secure IT Systems*. NordSec, October 2007.

[12] A. Suciu, D. Lebu, and K. Marton, "Unpredictable random number generator based on mobile sensors," in *Intelligent Computer Communication and Processing (ICCP), 2011 IEEE International Conference on*. IEEE, 2011, pp. 445–448.

[13] B. Sanguinetti, A. Martin, H. Zbinden, and N. Gisin, "Quantum random number generation on a mobile phone," *arXiv preprint arXiv:1405.0435*, 2014.

[14] Y. Ding, Z. Peng, and C. Zhang, "Android Low Entropy Demystified," in *Proceedings of the IEEE International Conference on Communications (ICC)*. IEEE, June 2014, pp. 1–6.

[15] J. Voris, N. Saxena, and T. Halevi, "Accelerometers and Randomness: Perfect Together," in *Proceedings of the Fourth ACM Conference on Wireless Network Security*, ser. WiSec '11. New York, NY, USA: ACM, 2011, pp. 115–126. [Online]. Available: http://doi.acm.org/10.1145/1998412.1998433

[16] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," *National Institute of Standards and Technology*, vol. 800-22, no. 1a, pp. 1–131, April 2010.

[17] R. Ellis, *Entropy, Large Deviations, and Statistical Mechanics*, first edition ed. Springer, 1985.

[18] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator," in *Selected Areas in Cryptography*. Springer, 1999, pp. 13–33.

[19] R. McEvoy, J. Curran, P. Cotter, and C. Murphy, "Fortuna: cryptographically secure pseudo-random number generation in software and hardware," in *Irish Signals and Systems Conference, 2006. IET*. IET, 2006, pp. 457–462.

[20] S. Muller, "Cpu time jitter based non-physical true random number generator," *DOI= http://www. chronox. de/jent/doc/CPU-Jitter-NPTRNG. html*, 2013.

[21] "Haveged entropy gatherer. http://www.issihosts.com/haveged/."

[22] A. Francillon and C. Castelluccia, "TinyRNG: A cryptographic random number generator for wireless sensors network nodes," in *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops, 2007. WiOpt 2007. 5th International Symposium on*. IEEE, 2007, pp. 1–7.

[23] G. L. Re, F. Milazzo, and M. Ortolani, "Secure Random Number Generation in Wireless Sensor Networks," in *Proceedings of the 4th international conference on Security of information and networks*. ACM, November 2011, pp. 175–182.

[24] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergnaud, and D. Wichs, "Security Analysis of Pseudo-Random Number Generators with Input: /dev/random is not Robust," *International Association for Cryptologic Research*, pp. 1–31, 2013.

[25] Google, "Android developer documentation https://developer.android.com/guide/index.html."

[26] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing Speech from Gyroscope Signals," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 1053–1067. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/michalevsky

[27] N. D. Lane, Y. Chon, L. Zhou, Y. Zhang, F. Li, D. Kim, G. Ding, F. Zhao, and H. Cha, "Piggyback crowdsensing (pcs): Energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '13. New York, NY, USA: ACM, 2013, pp. 7:1–7:14. [Online]. Available: http://doi.acm.org/10.1145/2517351.2517372

[28] D. E. Eastlake, S. D. Crocker, and J. I. Schiller, "Randomness requirements for security," RFC 1750, Dec, Tech. Rep., 1994.

[29] Qualcomm, "Trepn Profiler. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler."

[30] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "DSP.Ear: Leveraging Co-processor Support for Continuous Audio Sensing on Smartphones," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '14. New York, NY, USA: ACM, 2014, pp. 295–309. [Online]. Available: http://doi.acm.org/10.1145/2668332.2668349

[31] G. Metri, W. Shi, and M. Brockmeyer, "Energy-Efficiency Comparison of Mobile Platforms and Applications: A Quantitative Approach," in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '15. New York, NY, USA: ACM, 2015, pp. 39–44. [Online]. Available: http://doi.acm.org/10.1145/2699343.2699358

[32] "F-droid. https://f-droid.org/."

[33] "k9mail application https://github.com/k9mail/k-9."

[34] "Keepassdroid application https://github.com/bpellin/keepassdroid."

[35] "Randommusicplayer https://github.com/android/platform_development/tree/master/samples/RandomMusicPlayer/src/com/example/android/musicplayer."

[36] "Addi application https://code.google.com/p/addi/."

[37] "Aagtl application http://aagtl.work.zoff.cc."

[38] "Android activity manger profiler shell commands http://developer.android.com/tools/help/shell.html."

[39] "Android `getevent` shell command https://source.android.com/devices/input/getevent.html."

[40] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *International Conference on Software Engineering (ICSE'13)*, 2013, pp. 72–81.

[41] L. Torvalds, "Random.c linux file https://github.com/torvalds/linux/blob/master/drivers/char/random.c."

[42] W. X. R. R. C. Sanorita Dey, Nirupam Roy and S. Nelakuditi, "AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable," in *Proceedings of NDSS' 14*. San Diego, CA, USA: ACM, February 2014.

[43] S. H. Kim, D. Han, and D. H. Lee, "Predictability of Android OpenSSL's Psuedo Random Number Generator," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, November 2013, pp. 659–668.

[44] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An Emprircal of Cryptographic Misuse in Android Applications," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, November 2013.

[45] H. Corrigan-Gibbs, W. Mu, D. Boneh, and B. Ford, "Ensuring High-quality Randomness in Cryptographic Key Generation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 685–696. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516680

[46] E. Barkan, E. Biham, and N. Keller, "Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication," *J. Cryptol.*, vol. 21, no. 3, pp. 392–429, Mar. 2008. [Online]. Available: http://dx.doi.org/10.1007/s00145-007-9001-y

[47] B. Sunar, W. J. Martin, and D. R. Stinson, "A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks," *IEEE Trans. Comput.*, vol. 56, no. 1, pp. 109–119, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1109/TC.2007.4

[48] A. Chefranov, S. A. Abhari, H. Alavizadeh, and M. F. Zanjani, "Secure True Random Number Generator in WLAN/LAN," in *Proceedings of the 6th International Conference on Security of Information and Networks*, ser. SIN '13. New York, NY, USA: ACM, 2013, pp. 331–335. [Online]. Available: http://doi.acm.org/10.1145/2523514.2527098

[49] C. Hennebert, H. Hossayni, and C. Lauradoux, "Entropy harvesting from physical sensors," in *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*. ACM, 2013, pp. 149–154.
[50] K. Michaelis, C. Meyer, and J. Schwenk, "Randomly Failed! The State of Randomness in Current Java Implementations," in *Proceedings of the 13th International Conference on Topics in Cryptology*, ser. CT-RSA'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 129–144. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36095-4_9
[51] V. Gaglio, A. De Paola, M. Ortolani, and G. Lo Re, "A TRNG Exploiting Multi-source Physical Data," in *Proceedings of the 6th ACM Workshop on QoS and Security for Wireless and Mobile Networks*, ser. Q2SWinet '10. New York, NY, USA: ACM, 2010, pp. 82–89. [Online]. Available: http://doi.acm.org/10.1145/1868630.1868646

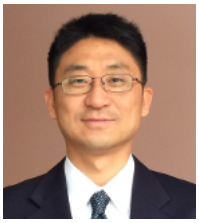
Kyle Wallace Kyle Wallace is a graduate student currently pursuing his Ph.D at The College of William and Mary. He received a B.S. in both Computer Science and Applied Discrete Mathematics from Virginia Tech in December 2012, and a M.S. in Computer Science from William and Mary in May 2015.

Kyle Wallace is Co-Advised by Dr. Gang Zhou and Dr. Kun Sun. His research interests include mobile computing, mobile security, entropy generation, sensor data analysis, and algorithm design.

Kevin Moran Kevin Moran is a graduate student in the Computer Science Department at the College of William and Mary pursuing a Ph.D degree. He graduated with a B.S. in Physics and Computer Science from the College of the Holy Cross in 2013, and an M.S. in Computer Science from the College of William and Mary in 2015. He is currently a member of the SEMERU research group and advised by Dr. Denys Poshyvanyk. His main research interest involves facilitating the processes of Software Engineering, Maintenance, and Evolution with a focus on mobile devices. Kevin was recently the second place winner among Graduate Students in the ACM Student Research Competition at ESEC/FSE'15.

Ed Novak Ed Novak is a sixth year graduate student pursing a Ph.D. in computer science at the College of William and Mary. Advised by Dr. Qun Li, he plans on graduating in August of 2016, after which he will join the faculty at Franklin and Marshall College. His research interests include cybersecurity and privacy on smart mobile devices and he recently won the award "Honorable Mention for Best Paper," for his submission at Ubicomp 2016. He received his M.S. in computer science from William and Mary in 2012 and his B.A. in computer science from Monmouth College in 2010.

Gang Zhou Dr. Gang Zhou is an Associate Professor, and also Graduate Director, in the Computer Science Department at the College of William and Mary. He received his Ph.D. degree from the University of Virginia in 2007. He has published over 70 academic papers in the areas of sensors and ubiquitous computing, mobile computing, body sensor networks, internet of things, and wireless networks. The total citations of his papers are more than 5000 according to Google Scholar, among which five of them have been transferred into patents and the MobiSys 2004 paper has been cited more than 800 times. He is currently serving in the Journal Editorial Board of IEEE Internet of Things as well as Elsevier Computer Networks. He received an award for his outstanding service to the IEEE Instrumentation and Measurement Society in 2008. He also won the Best Paper Award of IEEE ICNP 2010. He received NSF CAREER Award in 2013. He received a 2015 Plumeri Award for Faculty Excellence. He is a Senior Member of ACM and also a Senior Member of IEEE.



Kun Sun Dr. Kun Sun is an assistant professor in the Department of Computer Science at College of William and Mary. He received his Ph.D. in Computer Science from North Carolina State University in 2006. His research focuses on systems and network security. Dr. Sun has more than 10 years working experience in both industry and academia. Before joining W&M, he was a Research Professor in George Mason University. Before that, he was a Senior Research Scientist in Intelligent Automation Inc. at Rockville Maryland. He was a Member of the Technical Staff at Bell Labs, Lucent Technology in 2000. His current research focuses on trustworthy computing environment, moving target defense, smart phone security, and password management.