Understanding and Enriching Randomness Within Resource-Constrained Devices

Kyle Wallace
College of William and Mary - Arts & Sciences, kylemw@vt.edu

Understanding and Enriching Randomness Within Resource-Constrained Devices

Kyle M. Wallace

Williamsburg, Virginia

Bachelor of Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 2012
Master of Science, College of William & Mary, Williamsburg, Virginia, 2014

A Dissertation presented to the Graduate Faculty
of The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
August 2018

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

[illegible]

Kyle M. Wallace

Approved by the Committee, August 2018

[illegible]

Committee Chair
Associate Professor Gang Zhou, Computer Science
College of William & Mary

[illegible]

Associate Professor Kun Sun, Center for Secure Information Systems
George Mason University

Weizhen Mao

Professor Weizhen Mao, Computer Science
College of William & Mary

[illegible]

Associate Professor Peter Kemper, Computer Science
College of William & Mary

[illegible]

Professor Songqing Chen, Computer Science
George Mason University

COMPLIANCE PAGE

Research approved by

Protection of Human Subjects Committee

Protocol number(s):

PHSC-2016-02-01-10895-gzhou
PHSC-2017-09-27-12380-gzhou

Date(s) of approval:

2016-02-17 through 2017-02-17
2017-10-05 through 2018-10-05

ABSTRACT

Random Number Generators (RNG) find use throughout all applications of computing, from high level statistical modeling all the way down to essential security primitives. A significant amount of prior work has investigated this space, as a poorly performing generator can have significant impacts on algorithms that rely on it. However, recent explosive growth of the Internet of Things (IoT) has brought forth a class of devices for which common RNG algorithms may not provide an optimal solution. Furthermore, new hardware creates opportunities that have not yet been explored with these devices. In this Dissertation, we present research fostering deeper understanding of and enrichment of the state of randomness within the context of resource-constrained devices.

First, we present an exploratory study into methods of generating random numbers on devices with sensors. We perform a data collection study across 37 Android devices to determine how much random data is consumed, and which sensors are capable of producing sufficiently entropic 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 opportunistic collection of data from on-board sensors and applies a light-weight 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.

Second, we explore the design, implementation, and efficacy of a Collaborative and Distributed Entropy Transfer protocol (CADET), which explores moving random number generation from an individual task to a collaborative one. Through the sharing of excess random data, devices that are unable to meet their own needs can be aided by contributions from other devices. We implement and test a proof-of-concept version of CADET on a testbed of 49 Raspberry Pi 3B single-board computers, which have been underclocked to emulate resource-constrained devices. Through this, we evaluate and demonstrate the efficacy and baseline performance of remote entropy protocols of this type, as well as highlight remaining research questions and challenges.

Finally, we design and implement a system called RightNoise, which automatically profiles the RNG activity of a device by using techniques adapted from language modeling. First, by performing offline analysis, RightNoise is able to mine and reconstruct, in the context of a resource-constrained device, the structure of different activities from raw RNG access logs. After recovering these patterns, the device is able to profile its own behavior in real time. We give a thorough evaluation of the algorithms used in RightNoise and show that, with only five instances of each activity type per log, RightNoise is able to reconstruct the full set of activities with over 90% accuracy. Furthermore, classification is very quick, with an average speed of 0.1 seconds per block. We finish this work by discussing real world application scenarios for RightNoise.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I would like to extend my deepest gratitude and recognize the numerous people who offered their time, resources, and support throughout the years.

First, I thank my advisors, Dr. Gang Zhou and Dr. Kun Sun, for their constant feedback, guidance, insight, and support.

I recognize and thank those colleagues who have contributed to the work presented in this Dissertation: Dr. Kevin Moran, Dr. Ed Novak, Dr. Weizhen Mao, and Victoria Cooper.

Next, I would like to thank my current and past research colleagues, and all members of the LENS Lab research group.

Finally, I thank the faculty of our Computer Science Department, the Department Chair, Dr. Robert Michael Lewis, and the wonderful Computer Science administration team: Vanessa Godwin, Jacqulyn Johnson, and Dale Hayes.

To my family - Lester, Tammy, and Philip - for their unending support.

"…take what life has given you and do your best with it. I'm not going to say that anyone, that people are able to do anything. That's just empty words. But I will say that your limits are probably way farther out than you expect. And if you push yourself, you'll probably be really happy, and pleased, with the results."
-Clint 'halfcoordinated' Lexa

LIST OF TABLES

LIST OF FIGURES

Understanding and Enriching Randomness Within Resource-Constrained Devices

Chapter 1

Introduction

Random numbers and the generators thereof are a quietly essential part of the mainstream computing landscape [111, 87]. The values produced by an RNG are utilized in a wide variety of applications, ranging from facilitating a positive user experience all the way down to ensuring the core functionality of low-level algorithms. Random numbers help create the fun in games and gaming content (e.g. AI decision making, lotteries, procedural generation, initial configurations), aid in scientific computing (e.g. Monte Carlo methods, Markov models), assist with core OS and system functionality (e.g. stack pointer randomization, TCP random backoff), help keep communications and data secure (e.g. public and private cryptography schemes), and many other applications not listed here [87, 75, 96].

While random number generation is a topic that has been well studied in the context of traditional computing environments, the rapidly growing landscape of mobile and Internet of Things (IoT) devices has created a new space for research and exploration [43]. Mobile devices have proliferated and evolved into all-encompassing personal computers that not only perform familiar tasks, such as reading documents or browsing the web, 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 achieve greater awareness of their surroundings and become an interactive platform for data. This growing list of non-trivial use cases only adds to the argument that these devices must be treated less as secondary computational devices for users, and more as essential components of everyday life for those that choose to invest.

Unfortunately, while hardware continues to undergo rapid iteration and evolution, the algorithms running on these platforms may lag behind and miss opportunities to take advantage of new available features. This results in a mismatch of hardware and algorithm capability; in particular we focus on the subject of random number generation. On one hand, mobile devices now come equipped with a plethora of sensors designed to extract data from the environment around them. However, mainstream RNG implementations have failed to incorporate these new sources into their algorithms. Similarly, low-end IoT devices often come with constraints on the availability of computational resources, or ready access to sources of randomness that would be present in a desktop environment (e.g. user input). We find that this mismatch presents an opportunity to revisit the topic of random number generation within a new class of hardware. Finally, because of the uniquely single-purpose design of a vast majority of IoT devices, it remains unclear as to what the expected RNG behaviors and usage patterns in these devices are. This presents an opportunity for deeper research into understanding the structure of RNG use on these devices to better close the gap between theory and implementation.

1.1 Problem Statements

In this dissertation, we investigate the topics of cultivating better understanding of random number use and improving access to random numbers within the context of mobile and Internet of Things devices. Specifically, we approach the topic from three directions which try to take advantage of modern developments in hardware and software.

1) How well can new hardware sensors be leveraged to improve the production of random numbers on device?

2) How can we leverage the resources of multiple devices to help improve the quality of random numbers on resource-constrained devices?

3) What are the characteristics of RNG access patterns within the scope of resource-constrained devices?

Answers to these questions will aid in better understanding and development of random number generation algorithms specifically tailored for the devices that they run on.

1.1.1 Utilizing Hardware Sensors in RNG Algorithms

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 are capable of producing 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 light-weight 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.

1.1.2 Facilitating Collaborative Random Number Generation

Here, we lay the foundation for a collaborative framework to provide entropy on demand, which aims to move random number generation from an individual task to a collaborative one. By treating entropy as a shared resource, devices that are unable to meet their own needs can be aided by the devices in the network around them, facilitating improved performance for applications where randomness is required. We build and evaluate a proof-of-concept version of our protocol, called

CADET, on a testbed of Raspberry Pi 3B single-board computers, which have been under clocked to mirror the resource-constraints of IoT devices. Through this, we demonstrate the viability of the protocol as well as pave the way for future directions for research in this area.

1.1.3 Investigating and Profiling RNG Access Patterns

Our final work investigates and creates a link between device activities and RNG access patterns. To do this, we design and implement a system called RightNoise, which automatically profiles the RNG activity of a device by using techniques adapted from language modeling. First, by performing offline analysis, RightNoise is able to mine and reconstruct the structure of different IoT device activities from raw access logs. After recovering these patterns, the device is able to profile its own behavior in real time. We give a thorough evaluation of the algorithms used in RightNoise and show that, with only five instances of each activity type per log, RightNoise is able to reconstruct the full set of activities with over 90% accuracy. Furthermore, classification is very quick, with an average speed of 0.1 seconds per block.

1.2 Contributions

The overall result of this dissertation expands the scope of techniques and knowledge pertaining to random number generation and generators, particularly in devices with resource constraints.

Sensor-Based Random Number Generation for Mobile and IoT Devices. This work improves the production of random numbers on-device by utilizing new input sources. We first perform a data collection study on the current state of random data use and data different hardware sensors can provide. Based on our analysis of this data, we prototype a system to leverage the noise in hardware sensors for the production of random numbers. Our main contributions are:

- We conduct a data collection study surveying 37 Android devices of varying hardware capabilities. Our analysis of the data reveals two things: which sensors are suitable sources of random noise 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.

- 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.

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

- We discuss and provide insight into our findings, including the strengths and drawbacks of utilizing a sensor-based RNG in a mobile or IoT context.

Investigating a Collaborative and Distributed Entropy Transfer Protocol

This work moves random number generation from a strictly individual task to a collaborative one. We design and prototype a Collaborative and Distributed Entropy Transfer protocol (CADET) which collects randomness from participating devices in the system. The end result gives resource-constrained devices access to verified good entropy that can then be used to bolster the performance of their own RNG algorithms. Our main contributions are:

- To our knowledge, we propose the first general specification and implementation of an *open* distributed entropy transfer protocol, CADET, including details of the packet structure, device hierarchy, data flow, and core functionality.

- We provide a thorough evaluation of CADET, documenting its overall performance and overhead. We also provide insight into the design decisions, as well as investigate their effectiveness.

- We highlight critical results from the evaluation of the protocol in its current form, discussing its efficacy as well as paths for refinement and growth in future work.

Automatically Profiling IoT Device Activity by Characterizing RNG Usage This work investigates outside of the RNG algorithm itself and examines the actual patterns of requests made to the RNG module. We leverage the knowledge gained by our observations to build an activity profiling tool called RightNoise. RightNoise automatically analyzes RNG access logs during a particular system state and extracts meaningful patterns corresponding to different device activities. This activity profile can be used in a variety of ways, including detecting unwanted behavior. The main contributions of this work are:

- We perform a targeted user survey to understand the current state of IoT device adoption.
- We conduct the first (to the author's knowledge) investigation of RNG access patterns within the context of two specific devices, providing analysis of the underlying RNG activity structure.
- We design and implement a profiling tool called RightNoise, which automatically profiles the behavior of a system by analyzing RNG access patterns.
- We provide an evaluation of the various modules in RightNoise by using both simulated data and data collected from the devices examined in this work.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 presents relevant background on the subject of random number generation as well as a detailed survey of related work. In Chapter 3, we present our work on SensoRNG, a prototype random number generation algorithm for mobile and resource-constrained devices. SensoRNG utilizes on-board hardware sensors as input sources to the algorithm, helping improve generation on device. In Chapter 4, we treat random number generation as a collaborative problem and present our work on a Collaborative and Distributed Entropy Transfer (CADET) protocol. Chapter 5 presents our investigation into understanding the use of a RNG within resource-constrained devices, culminating with the presentation and evaluation of our profiling tool, Right-Noise. Finally, Chapter 6 summarizes the work presented in this dissertation.

Chapter 2

Related Work

Previous work related to this dissertation can be divided into several categories, including studies on existing RNGs, new methods for generating randomness, methods for remotely retrieving randomness, and the effects of low entropy RNGs.

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.* [44] in 2006 and Lacharme *et al.* [68] 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.* [21] 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 [68]. Our study finds that a single sensor sample is able to regularly provide much more. Another important feature of the LPRNG is the entropy estimation counter associated with each pool. 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 [22]. While our work does not explicitly investigate the security of PRNGs, we use these works as motivation for our exploratory study.

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 [60]. 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 [73]. 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 [51]. Other work has has suggested that CPU jitter could serve as a suitable entropy source for generating random numbers [80, 17]. 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 [31]. 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 [89]. Our primarily concern in our work is with randomness extracted from commodity sensors available in mobile and IoT devices. We use these approaches as motivation for choosing which hardware sensors to consider for analysis.

Sensor Randomness: The study carried out by Krkovjak *et al.* [66] investigates the microphone and camera in smart phones as promising sources of randomness. Similarly, Suciu *et al.* [98] 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.

External Entropy Retrieval: Closely related to this work is the idea of remote entropy retrieval. This was first realized with the introduction of `HotBits` in 1996 and `random.org` in 1998. These services provide on-demand random numbers drawn from radioactive and atmospheric noise respectively [108][87]. A patent in 2001 put forth the first concrete notion of *remote entropy*, describing the process of acquiring additional PRNG seeding information generated on remote servers and combining it with data already present locally [112]. However actual implementations of providing entropy on demand are still relatively new. The first attempt at this type of service came in 2012 by A. Toponce, who set up a single server for users to pull entropy from [100]. This was followed up by the National Institute of Standards and Technology (NIST) in 2013, who set up a beacon that broadcast 512 bit blocks of randomness every minute [82]. However, these bits were not intended to be used for security purposes. Note, that the NIST randomness beacon predates their Entropy as a Service (EaaS) proposal from 2015. In 2014, Canonical, the company behind Ubuntu Linux, announced the `pollenate` package. This was designed to help with reseeding the PRNG of Ubuntu virtual clients from a distributed network of servers which generated random strings [64]. Only in recent years has there been a growth of true EaaS services, such as netRandom [50]. Compared to these works, we differentiate ourselves in three key ways: 1) we collect excess entropy from participating devices in the protocol; 2) we specify and implement an open, lightweight distribution protocol; 3) our design is hardware agnostic (i.e., a software-only solution) and specifically accommodates resource-scarce devices; and 4) we provide a full evaluation of the performance of CADET.

Attacks on Low Entropy RNG States: A related subset of work involves attacks on PRNGs that are low on entropy. Of particular note are boot-time attacks, when entropy is expected to be the lowest due to the nature of how it is collected. It has been shown that this period of low entropy can lead to unfavorable outcomes such as factorable RSA keys [46], predictable TLS keys in virtual environments

[90], predictable OpenSSL keys on Android [62], predictable initial RNG outputs [28], and other yet undiscovered outcomes. These investigations motivate our work, which aims to provide entropy on demand to ensure the correct operation of any algorithm relying on random numbers.

Internet of Things Standardization: Recent efforts have tried to standardize software used across low-profile IoT devices to improve interoperability. Google has proposed their Android Things^{TM} platform as a standard executing environment for IoT devices [39]. Similarly, the Google Weave^{TM} communication protocol allows for these devices to more easily communicate through a unified language [41]. Other instances of unified protocols and platforms exist, such as Mozilla's *Things Gateway* [79], Open Habitat [29], or Home Assistant [7]. Should there be widespread adoption of a unified IoT platform or architecture standard in the future, this could pave the way for IoT devices to participate in a variety of useful distributed services, such as a distributed entropy system like CADET.

Intrusion Detection in IoT: A close class of work to ours is the idea of detecting anomalies in IoT devices, where an anomaly is any kind of unwanted behavior. Two closely intertwined categories for addressing this are intrusion detection systems (IDS) and behavior analysis. With regards to IDS, there have been a wide variety of approaches. Chen et. al propose using Complex Event Processing to determine the significance of different events in IoT real time [55]. Pacheco et. al present work on showing how a fusion between Intrusion Detection Systems and Anomaly Behavior Analysis can be combined to provide a security framework for smart architectures, like homes or buildings [83]. In a similar vein, Arrington et. al utilize a fusion of behavior analysis and intrusion detection, taking inspiration from immunity-based algorithms [6]. Amin et. al propose a hybrid signature-based and anomaly-based IDS system for sensor networks, called RIDES [2]. Many works have focused specifically on a class of ubiquitous computing devices running on 6LoWPAN technology. Kasinathan et. al has produced multiple works which construct a detection system on top of 6LoWPAN to bolster the security of IoT devices[58, 59]. Raza et. al proposed a system called SVELTE to mitigate several different attacks on 6LoWPAN IoT networks [88]. Le et. al consider the use of a topology attack against 6LoWPAN networks, and design an IDS system to defend against it. While we do not explicitly build an Intrusion Detection System in our work, the analysis we provide can be used to implement one by automatically classifying unrecognized system behavior at the RNG access level.

Behavioral Analysis of IoT: In the vein of behavioral analysis, several works discussed above utilize behavioral techniques in order to achieve their goal. In addition to that, Pacheco et. al present a method on securing IoT devices by applying the discrete wavelet transform to detect behavioral anomalies [84]. Hodo et. al use analysis of packet traces by an Artificial Neural Network (ANN) to thwart the effectiveness of Distributed Denial of Service attacks in IoT devices [48]. Zhou et. al considers the classification and analysis of different types of multimedia traffic, such as security camera video feeds, and uses the results to help secure the transmission of the data [115]. Yu et. al discuss at length potential methods for learning device behaviors in IoT in order to better mitigate attacks taking advantage of unpatched software [113]. Jia et. al create a platform called ContexIoT, which relies on application context in IoT devices to create a more robust permission system in the device[54]. Simpson et. al propose utilizing traffic analysis on an hub device or router in order to safeguard IoT devices with known vulnerabilities [94]. Our paper is most similar to these works in the fact that we utilize behavioral analysis. However, we are the first to utilize RNG access patterns as an information channel to classify IoT device behavior.

Chapter 3

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

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 are capable of producing 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 light-weight mixing algorithm to produce random numbers. We evaluate the quality of SensoRNG by using the National Institute of Standards and Technology (NIST) statistical testing suite, demonstrating that a sensor-based

RNG can provide high quality random numbers with only little additional overhead.

3.1 Introduction

Random numbers and the generators thereof are an essential part of the mainstream computing landscape [111, 87]. 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)[87, 75, 96].

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 [43]. 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 [107, 68]. 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 [44]. 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 [5, 51]. However it is impossible for legacy devices to take advantage of newer hardware. Furthermore, hardware is susceptible to problems such as bias, degradation, or back doors - all of which are typically more difficult to fix should they arise.

As a compromise between these two approaches, previous work has looked into extracting randomness from different sensors, such as the accelerometer or camera [66, 98]. However, these works are limited in their approach. Some are simply limited in the number of sensors they examine [66, 98, 93], 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 [66, 98, 93, 21]. Furthermore, very few works consider the overhead of using sensors as an input source in terms of power use and CPU overhead [66, 106].

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 sensor 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 of varying hardware capabilities. Our analysis of the data reveals two things: which sensors are suitable sources of random noise 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.

3.2 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 [91].

3.2.1 Entropy

Entropy is a standard metric in information theory that measures the uncertainty of events in a probability space [26]. 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.

3.2.2 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, random numbers see use in various constructs such as Address Space Layout Randomization (ASLR), stack canaries, establishing network connections, and much more.

While the applications of random numbers are relatively straightforward, the consequences of a poor RNG vary from application to application. For something as simple as a game of chance, it can simply lead to a poor user experience. In a scientific simulation, this can lead to lost time, or even indications of false trends within the data. On the other hand, a poor RNG feeding security algorithms can result in device vulnerability to attacks or data breaches.

3.2.3 The Linux PRNG

Figure 3.1 details the architecture of the Linux Pseudorandom Number Generator (LPRNG). The LPRNG draws its randomness from three main sources: user input (mouse and keyboard for desktops, touchscreen events for phones), interrupt request (IRQ) inter-arrival timings, and disk read/write timings. These events are collected into two pools and then 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

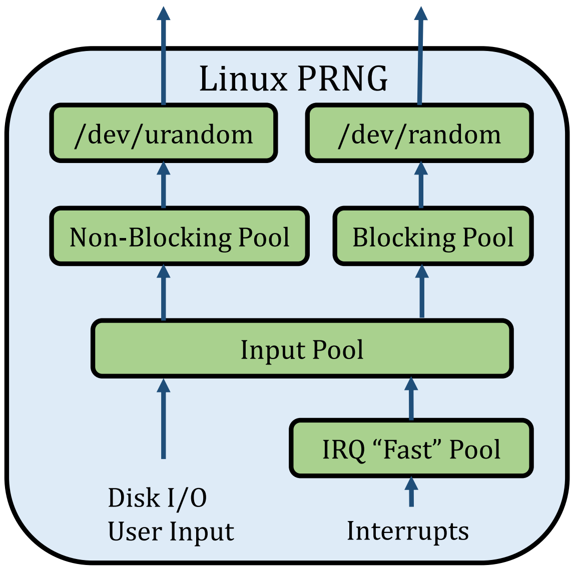


Figure 3.1: The Linux PRNG framework. User input events correspond to keyboard and mouse input, or user touch events for mobile devices.

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.

3.3 Data Collection Study

This section covers the details of our data collection study, in which we gather information about random number use and the different types of sensors in modern mobile and IoT devices. We target Android for ease of collection from a variety of sensors and devices, all of which run on top of the Linux kernel.

3.3.1 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. Because Android devices are produced by a number of manufacturers and span a wide range of capabilities, they are an ideal platform to explore the potential impacts of hardware diversity.

Sensor Name	Length (bits)	# Axes	Samples/second
Microphone	16	1	44100
Accelerometer	32	3	5
Magnetometer	32	3	5
Gyroscope	32	3	5
Radios	32	1	2
GPS	64	2	Variable
Camera	32	1	Variable

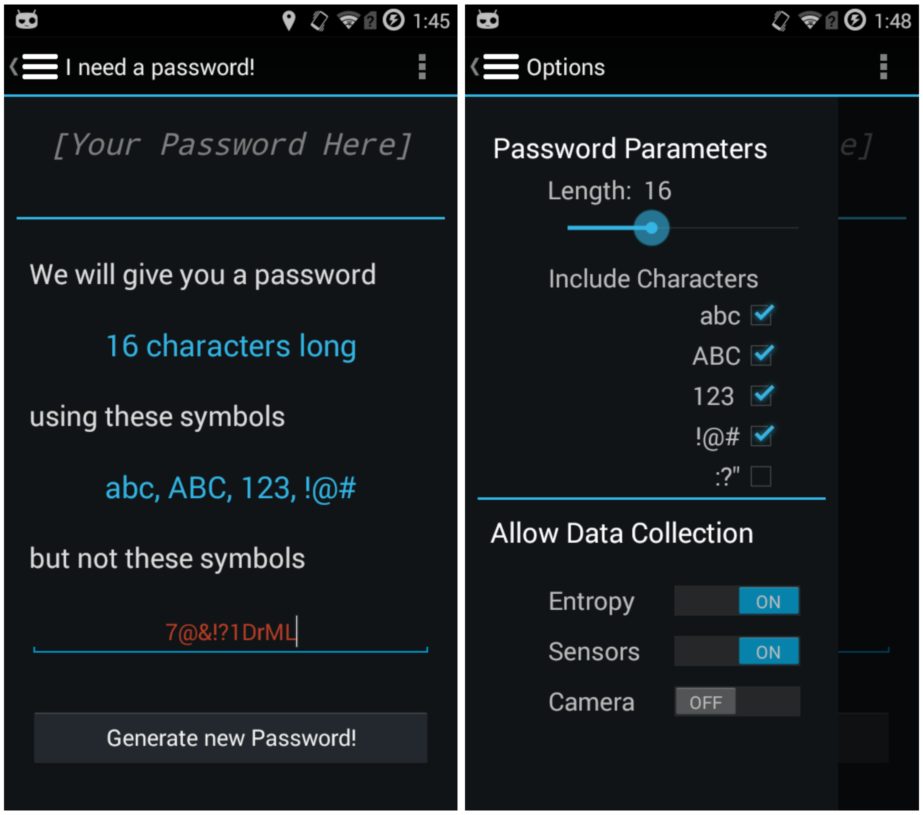
Table 3.1: Summary of the sensors chosen for study. GPS sample rate depends on movement, while camera sample rate depends on hardware.

Sensor Data: For our data collection study, we chose to include seven sensors commonly found in Android devices. Table 3.1 summarizes the sample size and rates for each sensor. Sensors with multiple axes (e.g., Accelerometer with x, y, and z directions) display the sample size for a single axis only. 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 [36].

Entropy Counter Data: The Linux PRNG tracks the amount of data available to the system when generating a random value. This amount is accessed through 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 stored in the main LPRNG input pool, and will increment and decrement accordingly when entropy is either added or removed. The maximum amount of random data that can be stored at any time is 4096 bits. We sample the entropy counter every 0.25 seconds.

3.3.2 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 3.2 shows two screens of the user front-end.



[Your Password Here]

We will give you a password
16 characters long
using these symbols
abc, ABC, 123, !@#
but not these symbols
7@&!?1DrML

Password Parameters
Length: 16

Include Characters
abc ✓
ABC ✓
123 ✓
!@# ✓
:?" ☐

Allow Data Collection
Entropy ON
Sensors ON
Camera OFF

Figure 3.2: Screenshots of the SensorPass application used for data collection.

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 method in which Android implements the camera API, it is only possible to gather camera data from the current active application screen. This is understandable from the standpoint of privacy, as malicious apps could covertly capture images or video without alerting the user being aware. Therefore, we rely on asking participants to manually collect camera data by using 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.

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 3.2 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 stationary user would only report one or two values.

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

Table 3.2: Summary of Sensor Data collected from SensorPass.

3.3.3 Analysis Methodology and Tools

Sensor Data: The main objective in analyzing the sensor data is to find where sufficient randomness can be extracted from the samples for further use. As illustrated in Figure 3.3, 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 tailored analysis for each sensor. This also eases the burden of processing when using the data in an RNG implementation. 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 the future.

For analyzing the randomness of a given stream, we utilize the NIST *Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications* [91]. The NIST suite is freely available to the public, open source, and provides a straightforward method 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*.

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

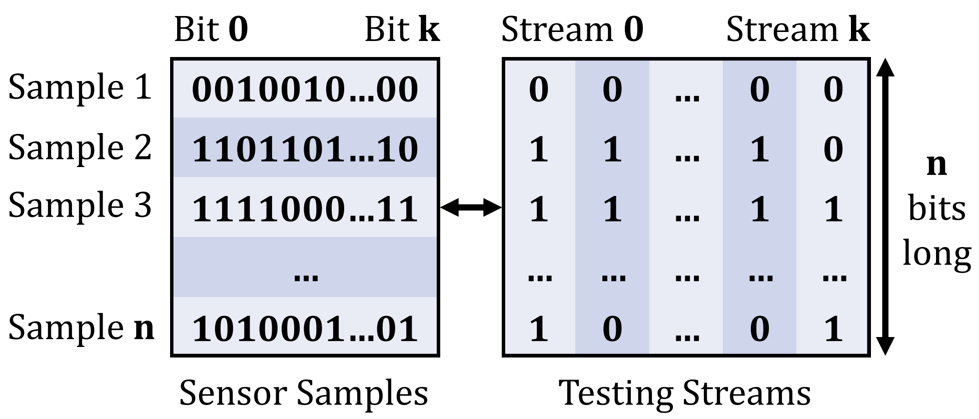


Figure 3.3: Diagram of the bitwise method used for analysis. The "Sensor Samples" block represents successive data samples from a sensor (horizontal), while the "Testing Streams" block represents the k individual streams (vertical) formed for analysis with the NIST suite.

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, discrete Fourier transform (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 behavioral aspect of randomness - for example, the rank test checks for periodicity in the data. Complete descriptions of each test and how to interpret the results can be found in the NIST suite documentation [91].

Entropy Data: Our main goal in analyzing the entropy counter traces is to assess the current demand for randomness 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 in the counter value.

3.4 Data Analysis Results

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

3.4.1 Sensor Data

We first present 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 approximately one usable bit of entropy)

Figure 3.4 illustrates the results of our analysis in a heat map. Note that some sensors included in the data collection study excluded due to their inability to provide any usable bits. We note that 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 methodology [66, 78, 93]. 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

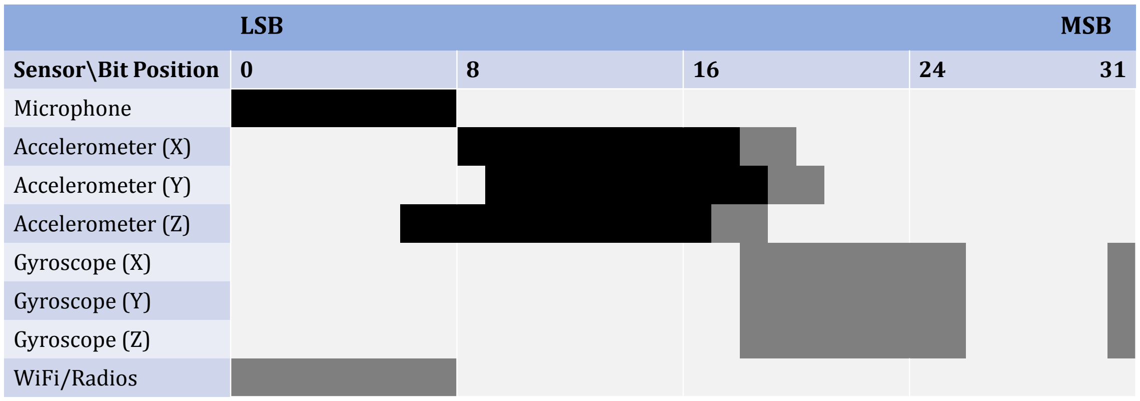


Figure 3.4: Heatmap of which bits from sensor samples show sufficient randomness. A black square indicates the bit is `good`, a gray square indicates a bit is `fair`, and an uncolored square indicates the bit is `poor`. We have excluded the Magnetometer, GPS, and Camera rows as they provided 0 good bits.

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 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.

3.4.2 Entropy Use

This section presents analysis of the entropy counter traces. Recall that the data collected for this part of the study consists of integer-valued time series with a sampling rate of 4 per second. Figures 3.5a-3.5d 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 dis-

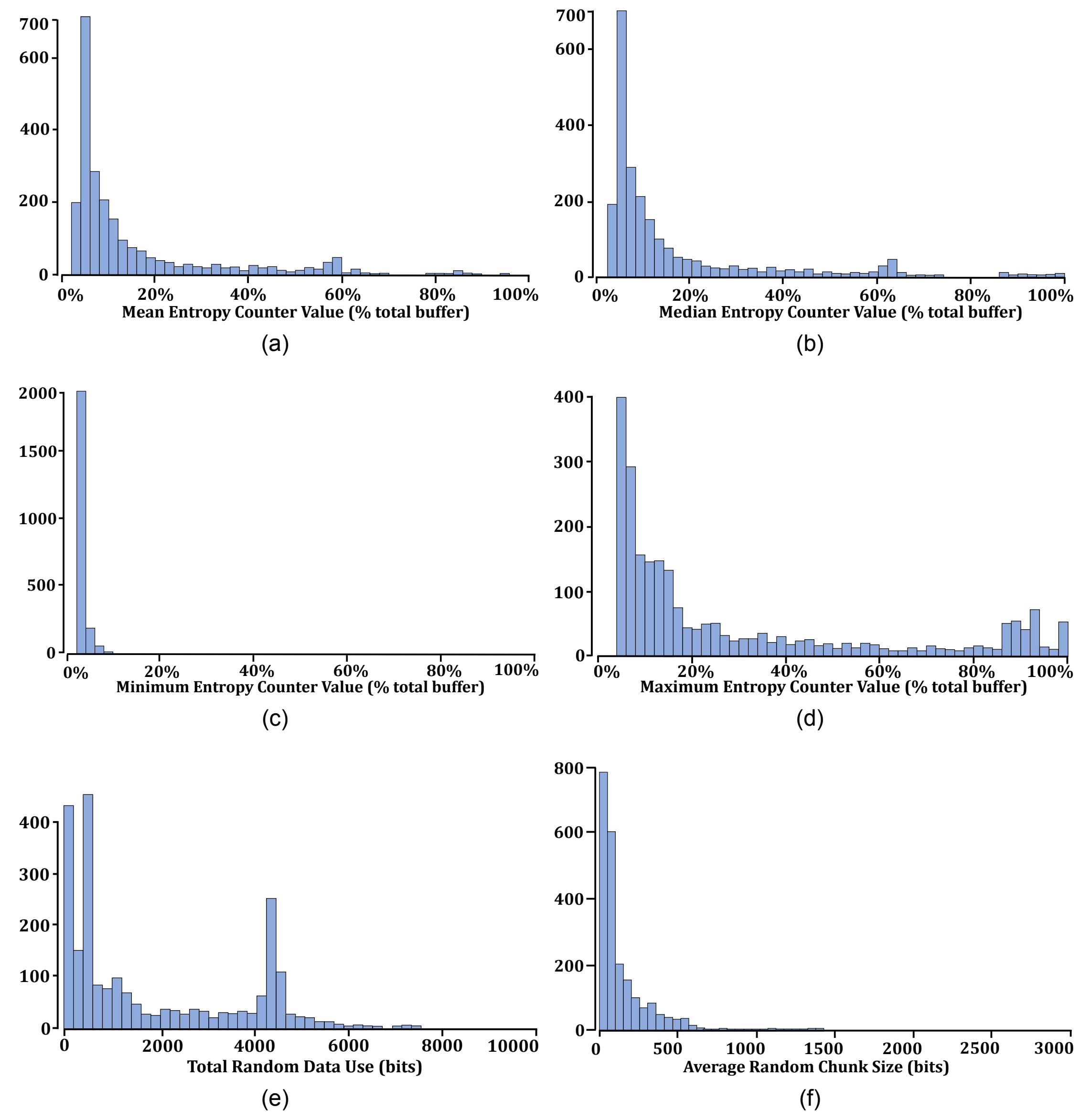


Figure 3.5: Histograms of various entropy use statistics. The Y axis is measured in number of traces. For 3.5a-3.5d, the X axis represents how full the buffer is (in percent). For 3.5e-3.5f, the X axis is measured in bits.

tribution, 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 3.3 further summarizes the quartiles of each statistic.

Total Entropy Use: Figure 3.5e illustrates the distribution of total random data use across all traces, while Table 3.4 summarizes basic statistics about the distribution. For a 3 minute trace, we calculate approximately 10 bits of randomness per second used on average, and less than 5.3 bits of randomness per second being used 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 approximately 4096 bits (the size of the APRNG buffer). However, we were unable to determine the cause of this phenomenon.

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 3.3: Quantiles of measured statistics across all traces. Values listed are in bits.

Magnitude of Use: Figure 3.5f 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 3.4 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 rare cases are larger requests made, but no request is able 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, periods

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

Table 3.4: Statistics of total random data use across all traces. Values listed are in bits.

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

3.5 SensoRNG

We now present the framework for SensoRNG, our proof-of-concept sensor-based RNG. Figure 3.6 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 enable opportunistic collection of sensor data without unnecessary polling overhead. Opportunistic collection has been utilized in other works to minimize the energy overhead of collection [69]. 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

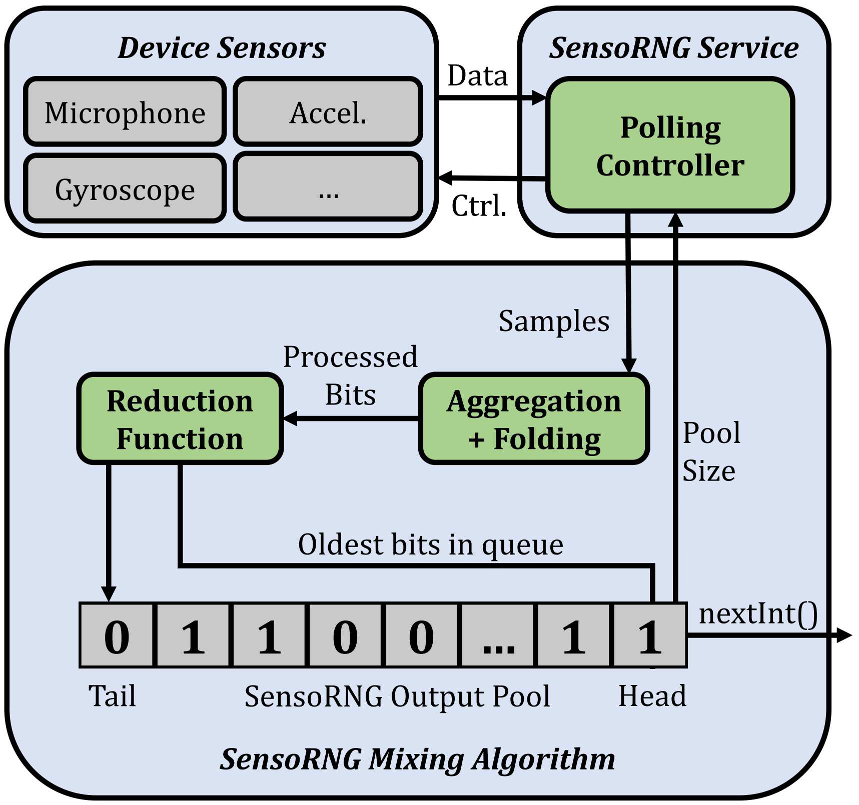


Figure 3.6: The SensoRNG Framework. Input is received from sensors via the polling controller and then queued for processing. Processed samples are merged with values already present in the buffer and then sent through a reduction function to further mix together temporally separate bits.

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 middle-man 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 ($E = G_1 G_2 \ldots G_k$) 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 operation

$$[(p+i) \text{ mod } l] = T[(p+i) \text{ mod } l] \oplus E[i] \tag{3.1}$$

Where $\oplus$ is bitwise xor. This process is repeated for a number of samples $E_1, E_2 \ldots E_n$. 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 ("s-box") to mix in key bits [24]. 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,

x	0000	0010	0100	0110	1000	1010	1100	1110
R(x)	0	1	00	01	10	11	0	1

x	0001	0011	0101	0111	1001	1011	1101	1111
R(x)	00	01	10	11	0	1	00	01

Table 3.5: Example substitution table `R(x)` in the reduction function with parameters $(n, m, r) = (4, 1, 2)$. Note that `x` is a fixed length binary input while `R(x)` ranges from 1-2 bits in length.

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.

An example substitution function `R(x)` is shown in Table 3.5. The function used in the SensoRNG algorithm is randomly generated 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 performed 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 $O(n)$.

3.6 SensoRNG Evaluation

In this section, we evaluate our prototype implementation of SensoRNG in comparison with the current Android OS random number generator `SecureRandom`.

3.6.1 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 3.3, we also include the cumulative sum, serial, and linear complexity tests [91]. 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 investigate two scenarios by simulating the statistical average and maximum random data usage found during our analysis in section 3.4. This is done by periodically making calls to the `getRandomBytes()` function at the rates that were determined in our data collection study - 10 bits per second for the average scenario and 55 bits per second at max.

To take power measurements, we utilize the Trepn power monitor for Qualcomm Snapdragon processors [86]. 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 [33, 76], and it has the ability to isolate and profile on a per application basis.

3.6.2 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 use a set of thresholds, below which different sensors will activate. 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.[1] 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.

[1]`https://developers.google.com/android/nexus/images` as of July 2018

3.6.3 Evaluation Results

We now present the results of our evaluation of SensoRNG in comparison to the APRNG's `SecureRandom`. Our evaluation hits on two key points: quality of the numbers produced, and power overhead of the SensoRNG algorithm.

	Nexus 4		
Test Name	SRNG (idle)	SRNG (typical)	S.Random
Frequency	0.3881	0.5955	0.9357
Block Frequency	0.0363	0.1718	0.5042
Cum. Sum (f)	0.5442	0.1969	0.9470
Cum. Sum (r)	0.2461	0.6838	0.4846
Runs	**0.0048**	0.0303	0.5442
Longest Run	0.9868	0.9681	0.8074
Rank	0.0302	0.0117	**0.0005**
FFT	0.7548	0.8676	0.2248
Approx. Entropy	0.2248	0.5697	0.4372
Serial (f)	0.9512	0.2622	0.8741
Serial (r)	0.9178	0.4465	0.9463
Linear Complexity	0.7695	0.3504	0.2248

	Nexus 5		
Test Name	S. Random	SRNG (idle)	SRNG (typical)
Frequency	0.9240	0.8255	0.7298
Block Frequency	0.3838	0.7981	0.6267
Cum. Sum (f)	0.7981	0.6786	0.2429
Cum. Sum (r)	0.6890	0.4654	0.7887
Runs	**0.0043**	0.0351	0.2968
Longest Run	0.5749	0.3627	0.8074
Rank	0.1188	0.3041	0.3669
FFT	0.0205	0.0965	0.3586
Approx. Entropy	0.0104	0.2133	**0.0007**
Serial (f)	0.0689	0.2077	0.7597
Serial (r)	0.6993	0.9733	0.8165
Linear Complexity	0.7791	0.3753	0.2429

Table 3.6: Comparison of reported p-values for **SensoRNG** (SRNG) and **SecureRandom** (S. Random) NIST suite results. A 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.

3.6.3.1 Quality

Table 3.6 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 [91].

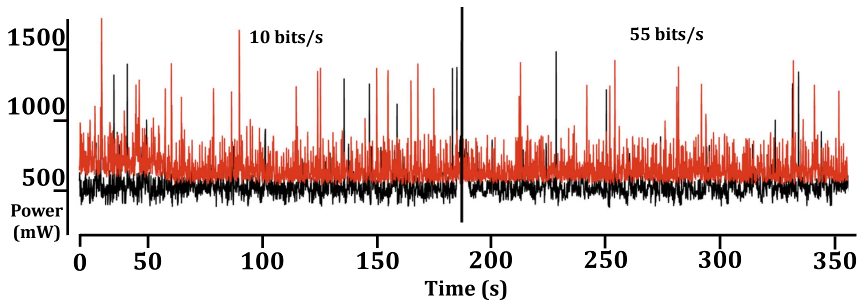
Overall, we find that SensoRNG performs favorably against `SecureRandom`. Both implementations pass all but one test, with a typical scenario passing all tests for SensoRNG. In terms of individual tests we find the results to be split evenly, with SensoRNG reporting higher p-values in some instances 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.

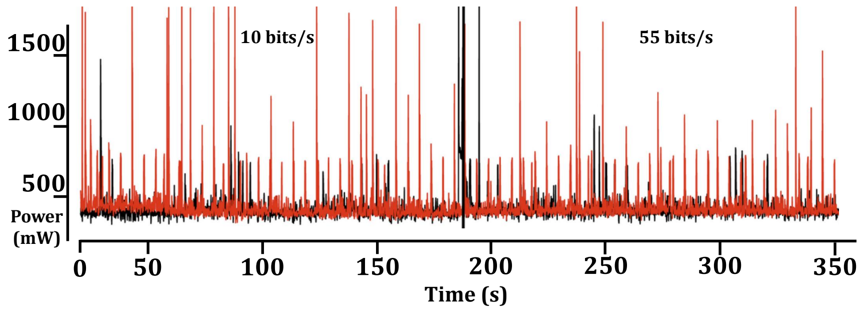
3.6.3.2 Power

Table 3.7 briefly summarizes the power draw for polling each sensor on each test device. The numbers were computed as follows: For each sensor we take a baseline measurement with no sensors for 3 minutes. 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



(a) Nexus 4



(b) Nexus 5

Figure 3.7: Power traces taken while producing random numbers. Black represents `SecureRandom` while red represents SensoRNG. The first 180 seconds show the average usage scenario (10 bits/sec), while the last 180 seconds show the max (55 bits/sec).

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

Table 3.7: 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`.

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.

Figures 3.7a and 3.7b 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 3.7 - that examines 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.

3.7 Applicability Study

To demonstrate the ability of the SensoRNG system to 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 [71], a 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.

3.7.1 SensoRNGLib Implementation

An important feature unable to be implemented in 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 facilitate thorough mixing. 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`.

3.7.2 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 Random Number generator (e.g. calls to `SecureRandom`). Thus, as our subject applications we used: k9 mail [56], keepassdroid [85], RandomMusicPlayer [35], Addi [12], and aagtl [116]. 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. Table 3.8 summarizes the results. Our experience indicates that modification of applications to utilize SensoRNGLib is very intuitive, requiring little effort on behalf of the developer even in more complicated applications.

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

Table 3.8: 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.

3.7.3 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) [37] 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 [38] 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 [34]. 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 [86]. 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 of this experiment are shown in Table 3.9. Overall, we find no significant deviation in cpu usage between the two implementations, suggesting that the SensoRNG does not impose additional computational overhead on applications themselves.

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

Table 3.9: Average Normalized CPU Usage for both the original and SensoRNGLib implementations of KeepassDroid and RandomMusic Player.

3.8 Discussion & Future Work

Our work has demonstrated the viability of utilizing sensors as a source of randomness. As the Internet of Things grows in scope, we can expect an increase in the number of low-profile devices dedicated to sensing and monitoring. For these devices, it may be the case that randomness can be more easily generated from sensor data rather than traditional methods. Future work could even target the sharing of this random data between IoT devices in local networks.

3.8.1 Limitations:

Sensor-based RNGs lack the ability to repeatedly generate a single 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 potential solution is to introduce a test mode which accepts input by reading from a single, predictable source, such as a file.

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.

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

Table 3.10: Comparison table for different SensoRNG implementations.

3.8.2 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 3.10 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.

3.8.3 Conclusion

This chapter presented 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, occasionally punctuated by short bursts of random data consumption. Furthermore, we show which sensors on mobile and IoT hardware are capable of meeting the demand for random data. To evaluate these claims we implement 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, utilizing small additional computational overhead when sensors are polled opportunistically, suggesting the viability of a fully optimized solution.

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

Chapter 4

CADET: Investigating a Collaborative and Distributed Entropy Transfer Protocol

The generation of random numbers has traditionally been a task confined to the bounds of a single piece of hardware. However, with the rapid growth and proliferation of resource-constrained devices in the Internet of Things (IoT), standard methods of generating randomness encounter barriers that can limit their effectiveness. In this work, we explore the design, implementation, and efficacy of a Collaborative and Distributed Entropy Transfer protocol (CADET), which aims to move random number generation from an individual task to a collaborative one. Through the sharing of excess random data, devices that are unable to meet their own needs can be aided by contributions from other devices. We implement and test a proof-of-concept version of CADET on a testbed of 49 Raspberry Pi 3B single-board computers, which have been underclocked to emulate the resource constraints of IoT devices. Through this, we evaluate and demonstrate the efficacy and baseline performance of remote entropy protocols of this type, as well as highlight remaining research questions and challenges in this area.

4.1 Introduction

In recent years, the concept of the *Internet of Things* (IoT) has materialized, encompassing a new class of computing hardware ranging from hobbyist boards, device prototypes, and flexible circuitry, all the way up to single board computers. While the data produced by some of these devices may be considered benign (e.g., a weather monitor), data from other devices may be cause for serious concern if accessed by unauthorized users. Particularly around the home, technology such as baby monitors, home security systems, home-assistants, and smart thermostats provide windows into a person's private life that a malicious entity may see as valuable targets. While IoT devices may be set up to utilize modern algorithms to protect these sources of sensitive data, the execution of these algorithms may be hampered by the capability of low profile hardware [110, 114].

One type of potentially affected algorithm is random number generation. Random Number Generators (RNGs) help facilitate the execution of many tasks across all areas of the computing hierarchy. The values produced are consumed by user-level applications such as games of chance and scientific simulation, but are also used in critical areas such as core OS systems, networking functionality, security algorithms, and many more. RNGs can be broadly categorized into two types: True Random Number Generators (TRNG) and Pseudo Random Number Generators (PRNG). A TRNG derives its values by sampling from some physical process that exhibits random tendencies [87], while a PRNG uses a combination of mathematical operations and initial "seed" data to produce a stream of statistically random values [68].

As it stands, random number generation is an individualized task. Standard computing environments typically employ a PRNG as their generator of choice to avoid the hardware costs of a TRNG. Computers, however, are deterministic environments, which makes finding suitable input sources for a PRNG a nontrivial task. Some PRNG implementations draw entropy (i.e., randomness) from the timing of different system events. The Linux PRNG, for example, uses disk I/O, interrupts requests (IRQs), and user input [68] [44]. While these events are readily available on a desktop computer or laptop, IoT and virtualized devices have created spaces devoid of user interaction. Similarly, due to the resource-constrained nature of IoT devices, the frequency of disk events is also greatly reduced or even completely absent. This combination of factors directly impacts the ability of the PRNG to gather sufficient entropy, which can lead to adverse affects such as boot-time entropy weakness, or extended periods of entropy starvation [105, 46].

Ideally, all devices would be able to take advantage of a hardware-based TRNG when needed. There has been some work in recent years to integrate TRNG capabilities directly into CPUs, such as Intel RDRAND for x86 [51]. Similarly, consumer devices effectively put a TRNG in a black box (e.g., USB stick, Smart Card) to augment on-board implementations [14, 103]. However, these newer hardware solutions are unavailable for devices without the necessary architecture or ports to utilize them (e.g., mobile phones, IoT devices, ARM-based devices). Similarly, purchasing new hardware for every device could be costly and time-consuming to implement and maintain, depending on scale (e.g., an office scenario). This problem is compounded for legacy or low cost devices, where hardware features may have been unavailable or omitted. Thus, for many devices in the IoT space, a software solution is the only answer.

While previous work has looked into improving PRNGs by better analyzing current sources of entropy or tapping into new ones (e.g., hardware sensors) [28, 90, 61, 42, 109], we instead turn our attention to augmenting the amount of data a device has access to. Specifically, we consider the idea that randomness can be treated a *shared resource*, where devices can export data that they are not using, and import additional data in times of high demand or low personal supply. In this way, random number generation is turned into a *collaborative task*. This is not the first time the idea of acquiring remote randomness has been explored. Websites have previously offered services employing randomness, such as shuffling lists or lotto drawings [87, 108]. Multiple patents discussing various mechanisms for distributing entropy have been submitted [112, 27, 81]. Most notably, a centralized entropy service was proposed by the National Institute of Standards and Technology (NIST) [105]. However, to the best of our knowledge, a functional framework and evaluation thereof has not been made publicly available.

Therefore, we further explore this idea and its efficacy by designing and creating a lightweight, flexible, and collaborative framework for devices to acquire randomness when needed. Our work is done with low profile IoT devices in mind, and we highlight the the following design choices and tradeoffs. First, we choose to effectively *crowdsource* (i.e., collect on a wide scale) the random data for this protocol from participating devices, rather than relying on specialized hardware located at centralized servers. This reduces the impact of individual hardware failure while also making the protocol capable of rapid deployment. Second, the framework is designed to be easily scaled to any scope, allowing both public and private instances (e.g., one single office building) to exist concurrently. Finally, we designed the protocol to be easy to access, and hardware agnostic. In this way, devices with very limited hardware or input methods are able to tap into the service without obtuse setup requirements or software.

In summary, the contributions of this work are as follows:

- To the the author's knowledge, we propose the first general specification and implementation of an open distributed entropy transfer protocol, CADET, including details of the packet structure, device hierarchy, data flow, and core functionality.
- We provide a thorough evaluation of CADET, including performance and overhead. We also provide insight into the design decisions, as well as investigate their effectiveness.
- We highlight critical results from the evaluation of the protocol in its current form, discussing its efficacy as well as paths for refinement and growth in future work.

4.2 CADET Overview

We present the overview of our remote entropy protocol, CADET. For our prototype implementation in this paper, the protocol exists at the application layer of

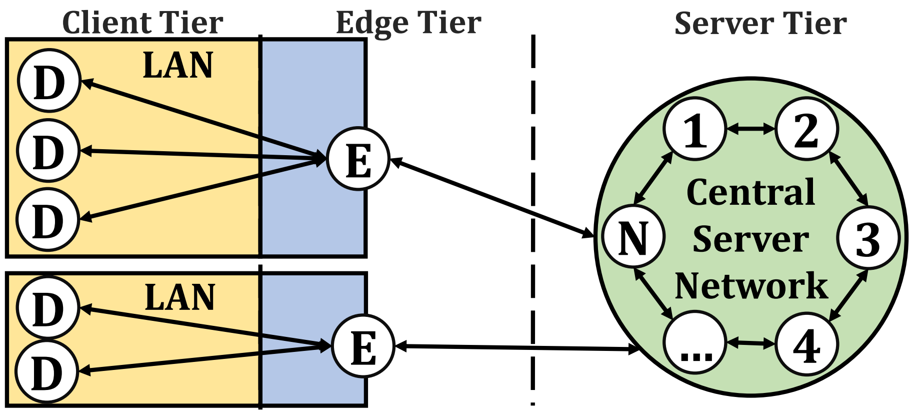


Figure 4.1: The CADET device topology. The server network is a collection of 1 to N devices which host entropy data. Edge nodes (E) bridge the gap between the local network (where client devices (D) are) and the server network.

the Internet stack. Our main goal is to offer two core functions for participants: the ability to contribute excess random data that they do not plan to use, and the ability for clients to request additional random data when needed. Through these, algorithms that rely on random numbers can be ensured a healthy supply of entropy, even on devices where harvesting randomness is a difficult task. Furthermore, we implement measures to enable secure data transfer for applications of a more sensitive nature, such as cryptographic key generation.

The CADET protocol is structured in a tree-like arrangement, distributed across three tiers - client, edge, and server. This style of construction takes advantage of the device hierarchy already seen in the Internet, where local devices connect through a gateway to access devices across the world. This topology is illustrated in Figure 4.1. We briefly discuss the device tiers and their purpose below.

Client: The lowest tier encompasses all devices on a local network (LAN). This is where both producers and consumers of entropy reside, including (but not limited to) laptops, smart phones, IoT devices, and virtual clients on servers. Devices in this tier will either upload excess data to the framework, or request additional data to consume.

Edge: The middle tier serves as a communication bridge between the client and server tiers. Logically, the edge consists of one device which serves as the gateway to the Internet at the edge of a LAN (e.g., a wireless access point or router). However, it could also be one designated device in the client tier, such as a home server.

Server: The upper tier is the network of central servers, which can range from simple desktop computers to rack servers. This tier is responsible for the heavy processing and bulk storage of data, as well as ensuring that requests from edge devices are met quickly and with quality output.

Data flows in two directions: from client to server (an upload of entropy into the service), or from server to client (a request for entropy from a client). We organize our discussion of how CADET accomplishes these tasks according to the design challenges for the protocol. These are data transport (how the data should flow through the service), data quality (how do we ensure the data is good), and data security (making the protocol robust against malicious entities). To that end, we formulate the following research questions to motivate our design over the course of this work:

RQ1) *How can entropy data be effectively collected and distributed between producing and consuming devices?* (§4.3)

RQ2) *How does the system need to react to varying entropy supply and demand to ensure correct operation?* (§4.3)

RQ3) *How and where can we verify that data being exchanged is of desired quality without sacrificing efficiency?* (§4.4)

RQ4) *How and where can security primitives be implemented to facilitate secure exchange without imposing excessive overhead?* (§4.5)

4.3 CADET Data Transport

Data Transport encompasses the flow of entropy data between devices in the CADET framework. In this section, we discuss the high level design of how uploads and requests are handled in the system, and introduce the various components used throughout both processes. Figure 4.2 illustrates the data flow architecture.

4.3.1 CADET Transport Design

As discussed in Section 4.2, devices participating are organized in a distributed, tree-like hierarchy, mimicking that of the Internet as a whole. By utilizing this structure, we aim to distribute the points of failure while still maintaining an ordered structure. A distributed service deals with the load balancing problem by moving a bulk of the collection work and initial processing out of the server tier and into the edge tier where there are more devices. The edge device for a given network serves as a staging ground for a local cache, similar to DNS. Each edge node keeps a small buffer of data available for local devices so that queries can resolve without traveling to the server level. As the edge device is both closer in a network sense and a physical sense, this reduces both transmission time of any packets, as well as the probability of network interference.

Data Upload: The top half of Figure 4.2 illustrates the flow of entropy data from clients to the server tier, while Figure 4.3a diagrams the corresponding packet exchange. Data is uploaded by client devices to their local edge node (1). Here, incoming data packets are collected and serialized by the packet processor mod-

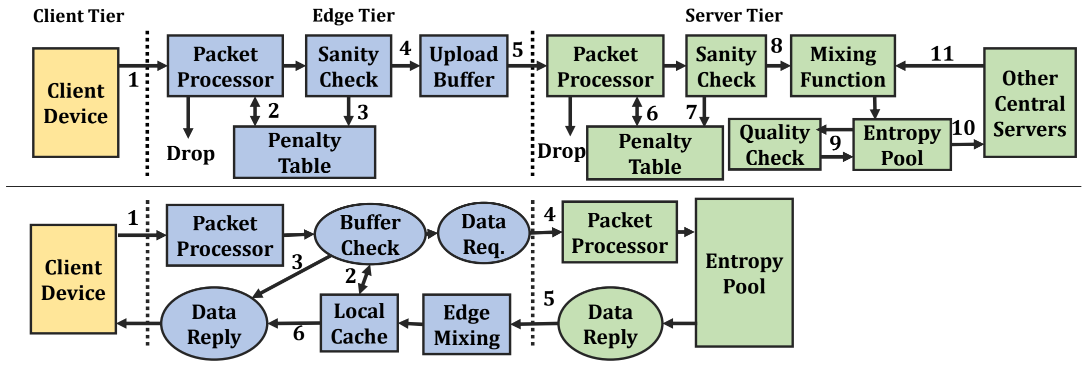
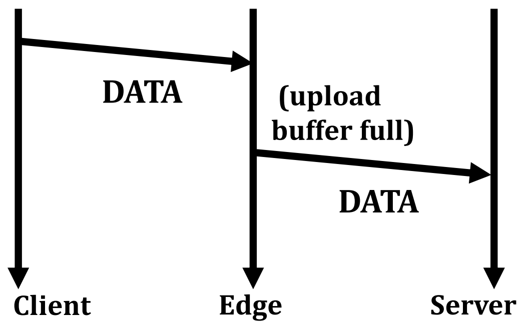
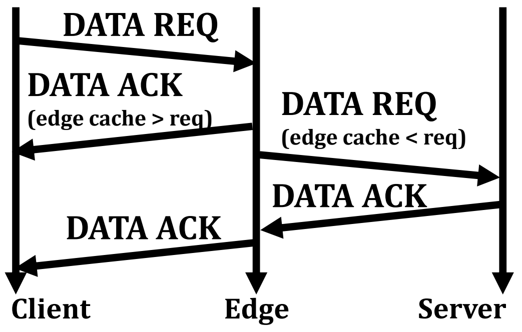


Figure 4.2: The CADET protocol data flow. The top half represents the upstream flow, with data going from client devices to the server tier. The bottom half represents the downstream flow, with data coming from the server tier down to client devices.



(a) Data Upload



(b) Data Request

Figure 4.3: Basic packet exchange timelines for data uploads and data requests in the CADET framework.

ule. If the client is in bad standing because of previous bad behavior, the packet may be dropped (2), otherwise the data is checked for initial quality (3). Should the data pass, the payload (entropy data) is added to a local upload buffer (4). After enough entropy data has accumulated, the edge node forwards all accumulated uploads to the server tier (5). Incoming packets at the server tier are serialized by the packet processor in a fashion similar to the edge tier (6-7). After processing, the data makes its way to the mixing function (8), which combines the new input with data already in the data pool on the server. Occasionally, the nodes in the server tier will partially exchange pool data (10, 11). This facilitates further mixing of input from devices all across the client tier.

Data Request: The bottom half of Figure 4.2 illustrates when a client makes a request for additional entropy, while Figure 4.3b diagrams the packet exchange for this process. A client sends an entropy request their local edge node (1). The request packet is processed and the edge node performs a check against its own local entropy cache (2). If there is sufficient data, then the edge node responds to the request immediately with a data packet (3). Otherwise, the edge node forwards a request to the server tier to acquire data to both refill its cache and respond to the client (4). Once data is received (5), it is mixed into the edge's local cache. Afterward, the edge node sends the needed data pcaket in response to the client's request (6).

4.3.2 CADET Packet Structure

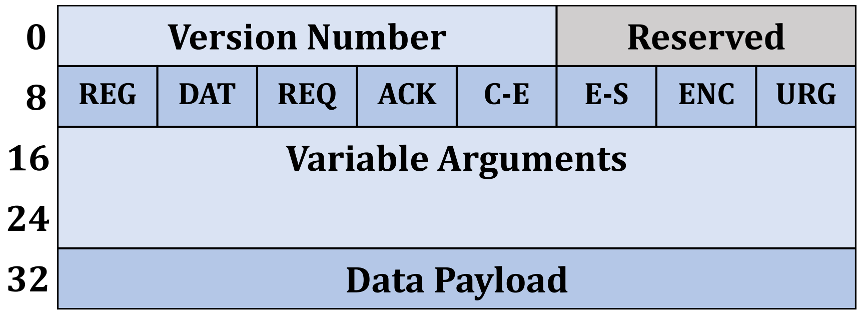


Figure 4.4: The CADET protocol packet structure. Each row is one byte (eight bits) long, except for the Data Payload section which is of a variable size.

Figure 4.4 illustrates the structure for all CADET protocol packets. The packet header is a four byte long chunk of information that describes important aspects about the type of action(s) instructed by the packet. There are three distinct chunks: protocol information, packet type and flags, and additional arguments. The first byte of the packet header is protocol information, where the first five bits are the protocol version number, while the last three bits serve to byte-align the header data. However, these bits could be used in future expansions of the protocol.

The second byte of the packet header specifies the packet type and various flags for the packet. The first two bits specify if the packet is a *registration* or *data* packet, while the second two bits specify whether the packet is a *request* or *acknowledgement*. The last four bits are flags which further specify the type of communication (i.e., whether it is *client-to-edge, edge-to-server*), whether the payload is *encrypted*, or if the packet is *urgent*, respectively. The third and fourth bytes of the packet header are reserved for additional arguments related to different packet types. Entropy request packets use the space to specify how large the request is (in bits), while entropy data packets use the space it to specify how large the contained entropy payload is (in bytes).

4.3.3 Data Availability

As the goal of CADET is to export a process that is performed on-device, care must be taken to minimize response time. Significant delays in delivery could impact a device's ability to properly run algorithms relying on random values. To address this issue, we implement a caching component ("local cache" or "edge cache") at the edge tier. This exploits the physical locality of edge devices (e.g., router) to mitigate network latency issues. Deciding on when to refill the cache depends on the supply and demand of the local network, and could potentially be modeled as a flow control problem. Deeper investigation of this topic has been left outside the scope of this paper. For our implementation, we instead use simple metrics. The maximum size of the buffer should be equal to 4096 bits (the typical size of a client's own randomness buffer), multiplied by the number of clients the edge is serving. This effectively reserves one buffer worth of data for each client. Meanwhile, the edge node should request additional data from the server tier when the cache reaches 25% capacity. These parameters mean that an edge node should always be ready to serve one quarter of its clients should demand spike.

There is also the possibility that a small number of clients could temporarily monopolize the local cache and impact the response time for other clients, causing local degradation of service. In consideration of this scenario, we implement a reserve-cache component for the caching mechanism at the edge tier. For this, we set aside a portion of the cache isolated from heavy users, should the edge not be able to adequately meet the demands of its heavier clients. To flag these heavy users, we implement a usage score based on the Exponentially-Weighted Moving Average (EWMA) formula. This is detailed in Equation 4.1.

$$US_t = usage_t + (decay * US_{t-1}) \tag{4.1}$$

US_t is a particular client's usage score at time t, and $usage_t$ is the client's current usage at time t. To be flexible with the speeds of different networks, t increments by one step every time a CADET packet is processed by the edge device. For a client to be considered a heavy user at time t, their current score must be above a given threshold. Our solution is inspired by the use of EWMA in TCP for congestion control [67]. Empirically, we choose a decay value of 0.96 and a threshold of 3 standard deviations above the mean usage score.

4.4 CADET Data Quality

Data quality refers to ensuring that any data transferred throughout the protocol eventually results in usable data for clients. We address the issue of data quality in CADET on three fronts as seen in Figure 4.2. With regards to input verification, we perform sanity checks on incoming packet payloads at both the edge and server tiers. For output verification, we periodically perform quality checks on the contents of the server pools to ensure outgoing data is sufficiently random. Finally, we ensure that data from all devices is thoroughly combined by basing the design of our mixing function on existing PRNG algorithms.

4.4.1 Sanity Checks

Sanity checks in CADET are intended to prevent excessive poor data from making it into the server pool. We introduce these checks in the packet processing phase at the edge and server tiers. When a device sends a data packet to the next tier, the contents are checked against a set of simple statistical properties (e.g., a balanced number of '0' bits and '1' bits). Depending on the outcome of the check, the data will either be forwarded to the internal data queue or discarded for being too low quality.

To quantify the problem of a device attempting to bulk upload bad data, the edge and central tiers maintain a penalty score for each uploading device. This score is based on the idea of a driver's license point system. Every time a driver gets a ticket, their license is assigned a certain number of points. After accumulating too many points (i.e., the driver is a 'bad' driver), their license is taken away. Similarly, when a device in CADET uploads poor quality data, points are assigned against the device. We summarize the general function of this penalty system in Figure 4.5. The number listed beneath the figure represent a user's penalty score, and increases left to right.

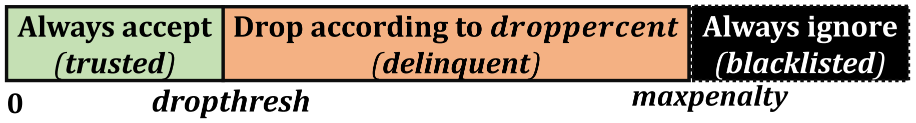


Figure 4.5: The CADET protocol drop strategy for sanity checks. User penalty increases left to right up to some threshold set at or above `maxpenalty`.

A device's penalty ranges from $[0, \infty)$. Points are removed or added depending on the quality of the data according to the penalty scheme. After a point threshold is reached, data upload packets are randomly ignored with a certain percentage until the device's penalty score reduces. This is to ensure that a device must always play fair to have points removed as they don't know whether a good or bad data packet will be ignored. Should the device continue to send bad data, all incoming data packets will be ignored and the device will be effectively blacklisted from participation. For the purposes of our prototype implementation, the values for `dropthresh` and `maxpenalty` are set to 10 and 35 respectively, while the formula for `drop_percent` is listed in equation 4.2.

$$drop\_percent = \frac{userpenalty - dropthresh}{maxpenalty - dropthresh} \tag{4.2}$$

Alternative equations for `drop_percent`, such as the sigmoid function, can also be used in order to avoid clients gaining a 100% drop rate. The base penalty scheme used in the prototype implementation of CADET is listed in Table 4.1, along with other alternative strategies, as different edge nodes may have different strictness requirements.

Num. Checks Passed	0/6	1/6	2/6	3/6	4/6	5/6	6/6
CADET Base	+5	+4	+3	+2	+1	0	-1
Loose	+4	+3	+2	+1	0	-1	-2
Strict	+10	+6	+3	+1	0	-1	-1

Table 4.1: Alternative sanity check penalty schemes for CADET.

To implement sanity checks in the CADET, we utilize a subset of 5 tests from the NIST suite, plus one test that compares current data against past data. Specifically, we use the frequency (Freq), runs, approximate entropy (AE), forward cumulative sum (CSum(F)), and reverse cumulative sum (CSum(R)) tests [91]. Each of these tests are computationally light, requiring only one or two passes through the data, keeping the amount of processing per bit linear.

4.4.2 Mixing Function

In the CADET architecture, the mixing function directly impacts output quality by how thoroughly it combines incoming bits. While any number of mixing algorithm designs could be created and implemented, we have drawn the design of our prototype mixing function from the Yarrow-160 PRNG [61]. Yarrow uses a two-pool system consisting of a fast pool and a slow pool, both of which accumulate entropy at different rates by alternating which pool is fed incoming input.

Using a similar design, we illustrate the mixing function used in CADET in Figure 4.6. As data enters the system, it accumulates in two pools (1). A majority of client input winds up in the fast pool, while periodically some input is diverted to the slow pool. Once a pool is full (2), its contents are concatenated with some of the oldest bits in the server's data pool (3). The combined data is hashed (4), and then reinserted at the tail of the buffer until data is requested (5). This process combines bits that are not temporally local, which helps keep the predictability of the pool low.

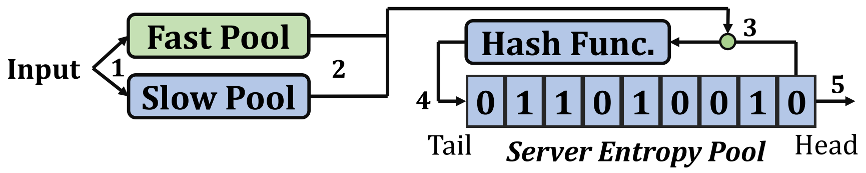


Figure 4.6: The CADET protocol mixing function, which serves to combine incoming data at the server level before storing it for future use.

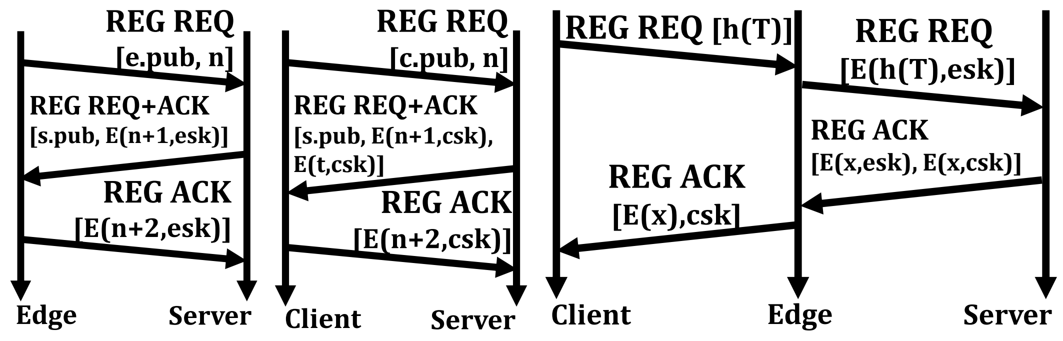
4.4.3 Quality Checks

The goal of quality checks is to ensure that the data that passes through the mixing function and is stored in a server pool is sufficiently random to be used by clients. To implement quality checks in CADET, we utilize a larger subset of statistical tests from the NIST suite [91]. This quality check is performed on the contents of the entropy pools located in the server tier to determine if the entropy eventually delivered to clients is sufficiently random. Depending on the power of the central server, more tests can be included in order to provide higher quality assurance, though care must be taken to avoid excessive computation which could impact response time.

4.5 CADET Data Security

Data security is the process of ensuring that a client's data or service quality is not affected by a malicious entity. This means both the contents of the data as well as the delivery of the data itself. We focus on three main threat vectors in the scope of this work - *service degradation*, *quality degradation*, and *eavesdropping*. Note that a service degradation attack in the context of CADET simply means that a client's request response times are significantly longer than expected. While previously mentioned components (e.g., usage score, sanity checks, mixing function) work together to mitigate degradation attacks, protecting against eavesdropping mandates the creation of secure communication channels between devices in the protocol.

To facilitate the creation of these channels, CADET implements a simple device registration component. While registration is not required for client devices to simply request entropy in the clear, it is a necessary step should the device wish to receive encrypted data. Both edge and client devices can register themselves, which establishes a secure channel between the device and the tier above it (i.e., client to edge, and edge to server). CADET's registration process is a hybrid of public key and token-based authentication in order to ease entropy consump-



(a) Edge Registration (b) Client Initialization (c) Client Reregistration

Figure 4.7: Packet exchange diagrams for the CADET registration process. **c, e, s** are shorthand for **client, edge, server**, respectively. Brackets represent packet payloads, comma separated. **x.pub** and **x.priv** refer to the public and private keys for a given device **x**. **n** is a nonce. **t** is a token. **E(d,k)** refers to encrypting data **d** under key **k**. **h** is a secure hash function. **esk, csk, cek** refer to shared keys between the two designated parties (e.g., esk is the edge-to-server key).

tion and computation on resource-constrained clients. For the purpose of our prototype, we have adapted a basic version of the Elliptic Curve Diffie-Hellman handshake algorithm to assist with registration.

4.5.1 Edge registration

For client devices to register to a CADET service, there must first be a registered edge node to communicate to. Thus, edge registration is regarded as the first step for allowing secure communication to occur. Figure 4.7a details the packet exchange for this process.

To act as an edge node, the device generates a new public-private key pair `(e.pub, e.pri)`, as well as a nonce `n`, and sends these to a server node (Packet 1). Once received, the server generates its own key pair `(s.pub, s.pri)`, and then computes a shared key `esk` based off of the received key `e.pub`. The server encrypts `n+1` under `esk`, and sends both its own public key `s.pub` and the encrypted nonce back to the edge node (Packet 2). The edge device can now compute `esk` and decrypt the nonce to verify the shared key. The edge node sends an encrypted `n+2` under the shared key to the server (Packet 3) which allows it to verify the shared key.

4.5.2 Initial Client Registration

Client registration is a less straightforward problem, as resource-constrained devices may have trouble generating the necessary randomness to repeatedly execute modern key exchange algorithms, or execute the algorithms in a timely fashion. Therefore, we have broken client registration into two parts - an initialization phase and a reregistration phase. Client initialization is a one-time execution of a key exchange algorithm to establish a shared key with a server, meaning that there is a one-time expense of entropy by the client. This process is roughly identical to edge registration, but also includes the exchange of a *token*. This registration token is used to help prove a client's identity for future client registration events. Figure 4.7b illustrates this packet exchange.

Similar to edge registration, a client first generates a fresh key pair `(c.pub, c.pri)`, a nonce `n`, and sends both pieces of information to a server node (Packet 1). The server generates a shared key `csk` from `c.pub` and encrypts `n+1` under `csk`. In addition, the server generates a token `t` (effectively a large chunk of random data) for the client device to facilitate future registration with edge nodes. The server sends its public key, encrypted nonce, and encrypted token to the client (Packet 2), where the client can also compute `csk` and verify the encrypted nonce. The client then responds to the server with an encrypted `n+2` so both parties can confirm that they have agreed on a shared key (Packet 3).

4.5.3 Client Reregistration

Once initial registration is completed, the client can register itself with the local edge node. This utilizes the token acquired from the initial registration step to avoid needing to do more than one key exchange. Whenever a client must register with any edge node, it can skip directly to this process instead of having to initialize once again. This avoids the situation where the client has to run a key exchange algorithm again, spending more entropy. Figure 4.7c illustrates this packet exchange.

The client takes its token `t` and the current time to make a tuple T, computes a hash `h(T)`, and sends it to the edge node it wants to register with (Packet 1). The edge node encrypts the token hash under its shared server key `esk` and forwards the registration request to the server tier (Packet 2). The server decrypts the hashed token and checks a local database to see if there is a match. If so, the server generates a new shared key `cek` for the client and edge to use. Two copies of this key are encrypted, one under the edge-server key `esk` and one under the client-server key `csk`. Both encrypted keys are sent back to the edge node (Packet 3). The edge then forwards the client its version of the encrypted key, after which both the edge and client devices decrypt their respective payloads and obtain the shared key `cek`.

4.6 Experimental Evaluation

To assess the capability and viability of the CADET protocol as described in this paper, we have implemented a prototype in Python. The following section details our evaluation of this implementation along several axes, including response time, overhead, performance, and security.

4.6.1 Testbed Setup

For all experiments in this paper, we utilize a testbed network of 49 Raspberry Pi 3B devices, all running the Debian-based Raspbian Jessie Lite OS [30]. The topology for this network is shown in Figure 4.8. For the client tier, 44 Pi devices are split into 4 networks of 11 nodes each, where each client and edge are connected via a single switch. The devices in each network act according to different sets of rules. Specifically, a *consumer network* consists of devices that will be mainly requesting entropy, a *producer network* consists of devices that will be mainly producing entropy, and a *balanced network* will have an approximately equal mix of consumers and producers. These networks attempt to model different ratios of producing devices to consuming devices.

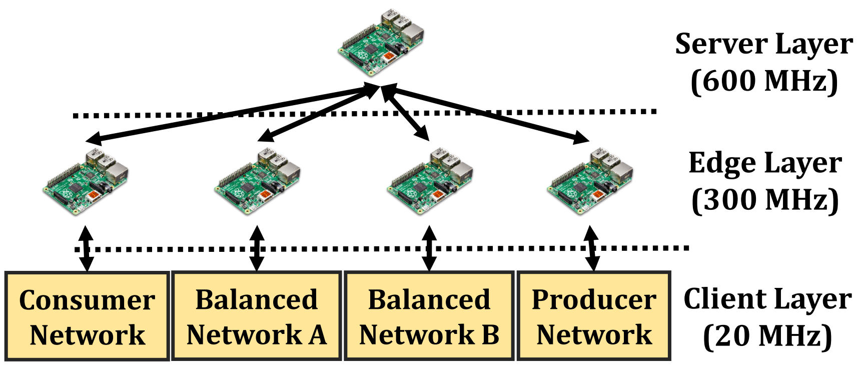
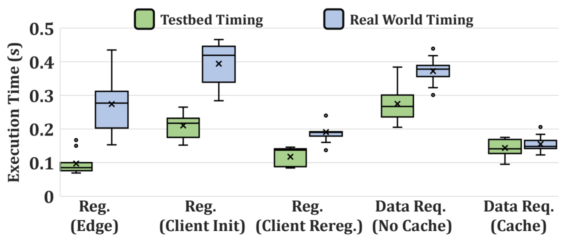


Figure 4.8: The Raspberry Pi testbed topology. Each network box in the client tier represents 11 Raspberry Pi devices under a particular edge node, connected via a single switch. The clock speed for each tier is listed beside the tier name.

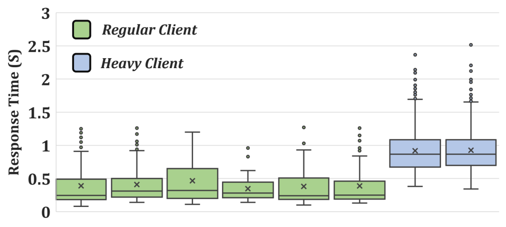
We have underclocked each Pi according to the labels in Figure 4.8, with the client tier operating at 20MHz with one core - the lowest stable speed. This is to emulate devices with processor constraints. While this is only one type of resource constraint, we find that the memory overhead of CADET is quite low, only requiring space to the client to store two encryption keys, their token, and the data that they request. Thus, the total memory footprint should stay under 4Kb for any device. For the edge tier, we choose 200MHz to mirror that of a low-end router. The server is at 600MHz, slightly under the speed of the original Raspberry Pi. For some experiments in this paper, we utilize a subset of the testbed in order to show data on one particular module. The code for CADET is written entirely in Python in around 1400 lines of code, utilizing UDP sockets to facilitate direct exchanges of data.

4.6.2 Data Transport

4.6.2.1 Protocol Timing



(a) Protocol Operations Timing



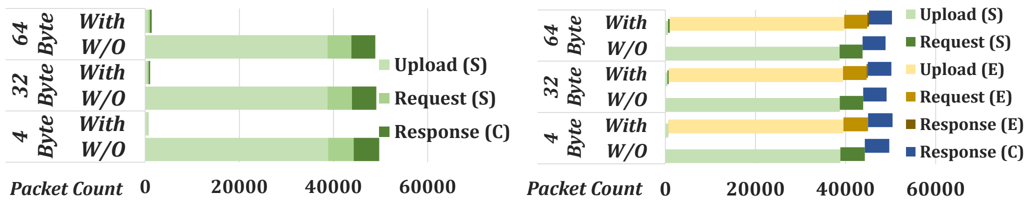
(b) Edge Response Time During Heavy Use

Figure 4.9: a) Distribution of protocol timings. The left and right boxes show the testbed (ideal) and real world timings respectively. Registration (Reg) of Edge (E), Client Init. (CI) and Client Rereg. (CR). Data Request (Data Req.) without (NC) and with (C) cache. b) Response time of the edge node to clients during periods of heavy use, in a network with six regular clients and two heavy clients.

We measure the time for a given process from the moment the first packet leaves the source device to the moment all processing for the final packet in the sequence has been resolved. The results of this experiment are summarized in Figure 4.9a. In general, average response times are very quick, below 0.25 seconds in all cases. With regards to registration, edge registration overhead is lower than client registration, likely due to the extra hop in the network and lower processing power of the client device. However, we highlight the fact that the average time for client reregistration is lower than that of initial client registration. This indicates that the token registration component for clients does indeed save time should the client device need to change edge nodes. With regards to the edge cache, the overhead difference is much more stark. On average, a client request experiences a 0.25 second response time when the edge node has no cache, but a 0.12 second response time when the cache can serve the request. These savings increase to almost a 0.3 second difference outside the testbed scenario where general Internet traffic and travel affects response times.

4.6.2.2 Edge Node Benefits

We generated 1000 random packets on each of the 43 client devices (43000 packets total, one device was malfunctioning) and tallied the number of packets processed by both the edge tier and the central tier. We do this for several configurations of upload payload sizes - small (4 bytes), medium (32 bytes) and large (64 bytes). Figures 4.10a and 4.10b summarize the data. As seen, introducing the edge node causes around a 98% drop in the number of packets the central server must process (4.10a), while the total number of packets sent within the system only increases around 3-5% due to the extra communication between edge and server (4.10b). These values are only expected to improve as the size of the edge tier grows. In the same vein, as the payload size increases, the number of data uploads from the edge tier to server tier increases as well. However, the increase is minor at best and is overshadowed by the savings on the server tier.



(a) Server Load Reduction by Edge (b) Network Cost of Edge

Figure 4.10: a,b) Total traffic processed by the server tier and on total network traffic sent with and without (W/O) the edge node. Values represent packets received by the (S)erver, (E)dge, or (C)lient. Data size (e.g., "4 Byte") is the average data chunk size uploaded by clients.

4.6.2.3 Usage Score

We orchestrated one network of 8 Raspberry Pis to investigate how well the usage score can identify heavy users. We plotted the usage score over time of all devices, two of which were intentionally tuned to be heavier users. Figure 4.11 shows the results of one data trace. While this is only one type of network, we see that the heavy users stay above the 'heavy user' threshold line between 60-80% of the time, while normal users are above the threshold only 5-15% of the time. For heavy users, it takes about 30-60 seconds to fall back beneath the heavy user threshold after finishing their period of increased use. Light users are much quicker, only taking around 5-10 seconds. This indicates that the user score does a good job of identifying heavy users quickly, without overly applying penalties. The decay factor and heavy user threshold can be tuned on a per-edge node basis. For example, lowering the decay factor will decrease the amount of time it takes for a user to transition from a heavy to normal user.

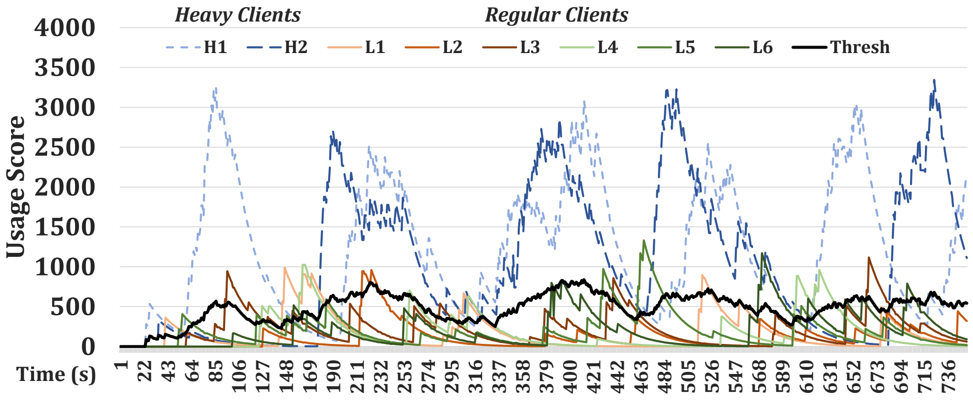


Figure 4.11: Usage score over time with a network of two heavy (dashed lines) users and six light (solid lines) users.

4.6.3 Data Quality

4.6.3.1 Sanity Checks

We investigated how the penalty system reacts when a client intentionally uploads a certain percentage of bad data (e.g., 5% of packets are intentionally poor). Note, that an honest client will statistically upload 1% bad data. Figure 4.12 shows these results. Under the default CADET penalty scheme, a client's penalty does not climb above the drop threshold of 10 points until 5% bad data, and clients do not have a high probability of being blacklisted until around 9% bad data. By implementing different penalty schemes (as discussed in Section 4.4), it is possible to push these numbers higher or lower on a per-edge node basis.

Table 4.2 summarizes how well the sanity checks perform in terms of classifying incoming data. Good data packets should be let through, while bad data packets (score ≤ 3 checks passed) should be dropped. For clients who upload

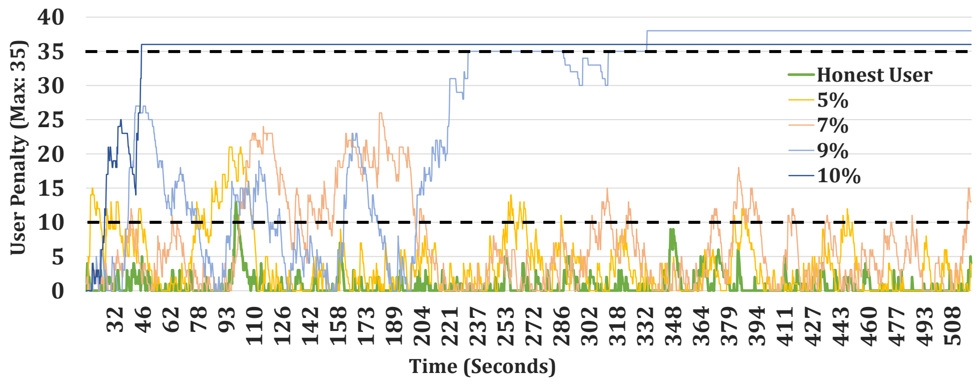


Figure 4.12: Plot of user penalty vs. time. Each entry represents the percent of uploads that are intentionally bad data. Drop threshold is marked at 10.

less than 5% bad data, we see that the classification error (FN + FP) stays under 2%. This number stays under 6% as the client bad data percent climbs to 8%, but quickly grows afterward. This represents a client's penalty score growing to the point that too many good packets are being dropped. As seen with Figure 4.12, the CADET penalty scheme tolerates up to around 8-9% bad data. This translates to at least 94% accuracy for the sanity checks. On a machine clocked at 300MHz, the current set of sanity checks take approximately 70-80ms to run on a data block size of 256 bits.

Client Behavior	Honest	2%	4%	6%	8%	10%
True Positive	98.76	97.44	95.42	93.08	90.16	84.54
True Negative	0	1.06	2.08	3.62	4.36	0.96
False Positive	0	0.88	1.72	2.48	4.26	8.94
False Negative	1.24	0.62	0.78	0.82	1.22	5.56
Accuracy	98.76	98.50	97.50	96.70	94.52	85.50

Table 4.2: Impact of client behavior on the performance of sanity checks.

4.6.3.2 Quality Checks

To evaluate the quality of the mixing function, we use the NIST statistical test suite on the data that accumulates in the server pool. Specifically, we allow 50000 bits to accumulate before running the tests. This process is repeated 200 times. The NIST suite documentation details how to calculate and interpret the p-values

	Freq.	B.Freq	CS(F)	CS(R)	Runs	LROO	AE
CADET	0.49	0.39	0.90	0.04	0.82	0.10	0.10
LPRNG	0.73	0.62	0.57	0.72	0.51	0.27	0.03

Table 4.3: Reported p-values for quality assurance tests, compared to the Linux PRNG.

for each test, but in general a higher value indicates a stronger suggestion of randomness, and p must be above 0.01 [91]. For comparison, we show our values against those produced by running the suite on the Linux PRNG [68]. Table 4.3 summarizes the results. Overall, we find that the values returned by CADET are comparable in quality to the LPRNG, as all tests are passed, and CADET shows stronger values on half of the tests. However, it is recommended by NIST that any values acquired from a remote entropy service be used to bolster the on-board RNG for a given device, rather than used directly [105].

4.6.4 Data Security

We use this section to highlight some of the more obvious threat vectors against CADET and how we mitigate them. We assume that all participating devices in the protocol are not compromised - an attacker does not have control of, or the ability to read data within a device. However, they are able to read any data moving between devices. We consider three different threat models in this section: eavesdropping, service degradation, and randomness degradation.

4.6.4.1 Eavesdropping

An eavesdropping attack occurs when an entity listens to data flowing between two devices. An attacker in this scenario wins if he is able to snoop on any data that he was not intended to receive. For the sake of argument, we only focus on encrypted data. Because of the registration process, all data flowing between devices is encrypted on every link. Therefore, an attacker's best chance is to deduce the shared key between devices during the registration phase.

The edge registration and client initialization steps currently both use the curve25519 Diffie-Hellman key-exchange algorithm, which has been shown to be both fast and secure [9, 23]. This means our security falls to the security of the algorithm. We therefore reasonably assume that edge-to-server communication and client-to-server communication is as secure as curve25519. When the client attempts to register with the edge node, their token is securely hashed before being sent. Thus, even though the hash is sent in the clear, the security of this step is on the strength of the hashing algorithm. Furthermore, even though an attacker can get a client's token hash, he does not have access to the client-server shared key, and therefore cannot decrypt the client-edge key. All other steps of the reregistration phase take place across secured lines (edge$\rightarrow$ server, server$\rightarrow$ client). Therefore, we conclude that the CADET protocol is robust against eavesdropping.

4.6.4.2 Service Degradation

A Service Degradation attack occurs when an entity attempts to affect the ability of other devices to properly participate in the CADET protocol. Proper participation, in the view of an honest client, is the ability to receive good data in a timely fashion. Therefore, an attack is considered successful if a client receives bad data, or if a client is sufficiently delayed in receiving data. Note, that we do not address what would be considered a standard "Denial of Service" attack, where an edge or server node is overwhelmed with traffic, as that is outside the scope of this work.

To cause a client to receive bad data, an attacker would have to upload enough data into the system to dilute the server entropy pool. However, three factors prevent the pool from being flooded. First, collecting data from many devices means that a single malicious device is greatly outnumbered by honest devices. Second, the mixing function at the server tier blends together data multiple sources, masking poor uploads. Finally, the sanity checks at both the edge and server tiers will quickly catch a malicious client and prevent it from uploading poor quality data in bulk.

To impact a client's response time, an attacker would need to continually drain the local cache at the targeted edge node. However, we easily address this by implementing the usage score, separating clients into heavy and regular users based on their recent request volume. When an edge cache is emptied, regular users have their requests answered by the reserved portion of the edge cache. As seen in figure 4.9b, we are able to keep the response time within the expected measurement average of 0.25s, even while the normal portion of the cache has been emptied. While there are more outliers, we attribute this to the larger number of packets being processed.

4.6.4.3 Randomness Degradation

A randomness degradation attack occurs when a large number of devices attempt to influence the quality of the service by bulk uploading known data. The aim is to make the eventual client output more predictable based on knowing or controlling a large majority of the input. This is similar in style to how a bot net would operate to negatively influence some service. While outright preventing bot net attacks is beyond the scope of this paper, we argue that CADET is resilient to this flooding style attack. There are two types of data that a malicious entity can upload - poor quality data and good quality data. We have already demonstrated that poor quality data cannot be uploaded in bulk due to the sanity checks at both the edge and server tiers. Therefore, we only worry about the scenario where a large number of attackers are uploading known data that passes the sanity check phase

We note that *all* client data at a particular edge is aggregated and serialized into a single large payload before being uploaded to a server node. This means a single benign client uploading data will reduce the effectiveness of the attack, as his data will be randomly added into the malicious payload. This process repeats at the server level, as incoming payloads from all edge nodes are combined into one main buffer. Even if we make the assumption that multiple edge nodes are uploading predictable data, the strength of the server mixing function also comes into play. By utilizing a two-pool design, and mixing back in data that is already in the randomness buffer, we introduce a high degree of nonlinearity that is drawn from the unpredictability of client request timings, which cycles data out of the buffer.

Some simple changes to the upload pipeline could also help further mitigate the effectiveness of this attack. First, the edge node can *require* data from multiple clients before uploading the aggregate payload. The edge (and server) could further measure some local sources of entropy, such as CADET packet inter-arrival times, and inject these bits between payload contributions from clients. Finally, the mixing function can be adjusted to require contributions from multiple edge nodes before insertion into the main buffer.

4.7 Discussion and Future Work

In this work, we have constructed a working prototype of the proposed Collaborative and Distributed Entropy Transfer protocol. However, there are many features and questions that are still open for exploration. We briefly summarize potential areas for exploration in further research below.

First is deeper analysis of supply and demand within CADET systems. As the number of devices within any particular instance increases, the load on the system becomes more complex and more difficult to predict. This may necessitate nodes in the system to adapt in a dynamic manner to ensure timely delivery of data. This could potentially be modeled as a flow control problem, but would require additional empirical data on the demands that a large scale system produces.

Next are questions surrounding the scope of deployment. In the current iteration of the protocol, we make the assumption that the edge node is a device that can be reasonably trusted (e.g., a home router). However, with mobile devices, scenarios where the user cannot trust the edge node will be much more common (e.g., public Wi-Fi at a local shop). Further investigation is needed to determine the amount of effort required to expand the protocol to cover these scenarios where trust may not be guaranteed.

Finally, we consider how to encourage participation in the CADET protocol. A collaborative solution is only as good as the data provided by participants. However, clients who contribute above a certain threshold may wish to be compensated for their resources. Similarly, building and maintaining a network of central servers requires hardware and bandwidth. We can imagine looking to potential incentive models from other industries, such as public utilities like electricity (e.g. users with solar panels), or sharing economy systems (e.g. ride sharing).

4.8 Conclusion

Random numbers power a wide variety of algorithms in modern computing, ranging from simulation to security. However, gathering the necessary entropy to ensure the correct operation of these algorithms has become a problematic task with the surge of resource-constrained devices in the Internet of Things. To alleviate this problem, we propose the initial designs for a Collaborative and Distributed Entropy Transfer (CADET) protocol, whereby devices that have generated an excess of entropy can indirectly assist those that are entropy deficient. Throughout this paper we have highlighted a number of design choices taken in order to maximize efficiency of the framework, utilizing a testbed of 49 Raspberry Pi 3B devices to gather additional supporting evidence. The groundwork has been laid for future work on this topic, with a number of open questions still remaining for exploration.

This work is partially supported by the U.S. Office of Naval Research under Grant N00014-16-1-3214 and N00014-16-1-3216, and by the U.S. National Science Foundation under grants CNS-1253506 (CAREER) and CNS-1618300.

Chapter 5

RightNoise: Automatically Profiling IoT Device Activity by Characterizing RNG Usage

The Internet of Things (IoT) encompasses the pervasiveness of technology in everyday life, with devices that sense, interact with, and control the world around them. However, due to the generally single-purpose nature of these devices, they often operate silently in the background, and problems that arise may go unnoticed for much longer periods of time than acceptable. To help address this problem, we propose a method of automatically characterizing device behavior by profiling Random Number Generator (RNG) access activity, the first investigation of this particular channel of information. Our work surveys and investigates the top two devices on the market - Google Home and a security camera setup - analyzing the behavior of the RNG while these devices are in use.

From this, we design and implement a system called RightNoise, which automatically profiles the RNG activity of a device by using techniques adapted from language modeling. First, by performing offline analysis, RightNoise is able to mine and reconstruct the structure of different IoT device activities from raw access logs. After recovering these patterns, the device is able to profile its own behavior in real time. We give a thorough evaluation of the algorithms used in

RightNoise and show that, with only five instances of each activity type per log, RightNoise is able to reconstruct the full set of activities with over 90% accuracy. Furthermore, classification is very quick, with an average speed of 0.1 seconds per block. We finish our work by discussing some immediate real world applications that RightNoise could be utilized in.

5.1 Introduction

Devices comprising the Internet of Things (IoT) have been integrated into all aspects of life, tasked with sensing, controlling, and enriching the world around them. A general design philosophy for IoT devices is to tailor the hardware and software to address a specific task or set of tasks. For example, in the idea of a smart home, some devices may sense the temperature, another may control the lighting, and a third may automatically inform residents that a member of the household is coming home from work. A byproduct of the single purpose nature of these devices is that they are often designed to be "set and forget". Once the device is installed and configured, they experience almost no further interaction unless something obviously wrong occurs. With declining costs for manufacturing and ease of distribution, the average number of these background devices one person owns has steadily risen.

A direct consequence of the nature of these devices is that there is a higher chance for unwanted activity to go unnoticed. While standard computing environments like laptops and desktops often see high degrees of user interaction, it is possible that some IoT devices may go several days or weeks without being touched at all. Therefore, if a device is malfunctioning, there is a good chance that it will go unnoticed for a longer period of time. Along with the growth in adoption of IoT devices has also come the unfortunate attention of malicious entities looking to take advantage of this new class of targets. Undesired events, such as hackers breaking into unsecured devices [95] or infected IoT devices being used in botnet attacks[4, 65, 3], are on the rise and have called into question the security of these devices as a whole.

In response to this growing spectrum of threats against an increasingly heterogeneous space of devices, researchers have proposed multiple solutions to better understand when an IoT device is exhibiting unwanted behavior. Stemming from research already present in networking, Intrusion Detection Systems (IDS) have been extended to attempt to cover the diversity of IoT devices [55, 6, 58, 83, 88, 59, 2, 70]. Other research has attempted to better understand and classify the behavior of IoT devices, both internally and externally, in order to detect deviations from expectation [6, 10, 84, 48]. As always with unwanted or malicious software, there is never one metaphorical silver bullet for prevention.

Our work takes inspiration from previous endeavors into device behavior analysis. We present the first investigation into a novel channel for profiling IoT device behavior by analyzing Random Number Generator (RNG) usage patterns. From our findings, we design a system called RightNoise, which automatically profiles the RNG access patterns of a system during a known configuration. RightNoise borrows concepts from language modeling in order to capture the behavior of the system under examination with over 90% accuracy. Once an activity profile is constructed, a system can monitor and report its activities in real time.

For the purpose of this work, we formulate and answer the following research questions.

RQ1) *What IoT devices have seen the greatest adoption with consumers?*

RQ2) *What is the structure of RNG access patterns in IoT devices?*

RQ3) *How can we best capture and express the structure of RNG access patterns?*

RQ4) *How well can device activities be inferred from RNG access patterns?*

RQ5) *How well can real time RNG access patterns be classified as according to known structure?*

To address RQ1, we conduct a user survey in which we pose questions about device ownership. To address RQ2, we use the results of our user survey and investigate the RNG behaviors of the most popular device setups. For RQ3, we use the results of our device investigations to formalize a method based on language modeling for capturing and expressing the RNG activity within a system. This method is implemented in a tool called RightNoise, which performs offline analysis to generate a grammar for an IoT device, and then conduct real time analysis to profile the systems behavior. We analyze the performance and accuracy of RightNoise to address RQ4 and RQ5.

Our contributions are as follows:

- We perform a targeted user survey to understand the current state of IoT device adoption.
- We conduct the first (to the author's knowledge) investigation of RNG access patterns within the context of two specific devices, providing analysis of the underlying RNG activity structure.
- We design and implement a profiling tool called RightNoise, which automatically profiles the behavior of a system by analyzing RNG access patterns.
- We provide an evaluation of the various modules in RightNoise by using both simulated data and data collected from the devices examined in this work.

We organize the rest of this chapter as follows. Section 5.2 presents our user study, in which we investigate what IoT devices people are actively adopting and using. Based on these results, we perform individual device studies in section 5.3 to determine the characteristics of RNG access patterns within the most popular devices. We go on to present the design and architecture of RightNoise in section 5.4 and 5.5, and subsequent evaluation of the system in section 5.6. We discuss potential applications of RightNoise in section 5.7 and future work in section 5.8.

5.2 User Questionnaire

To better select which consumer level devices to investigate during our study in Section 5.3, we first aim to understand the current state of IoT device adoption.

To do this, we conducted a user questionnaire, designed to help us better understand adoption of devices in the IoT landscape, as well as consumer sentiment towards IoT devices as whole. Answers were gathered between October 31st, 2017 and November 20th, 2017. In total we gathered responses from 203 unique participants across a variety of demographics. This user study was approved by the Institutional Review Board (IRB) at the author's institution. Participants were aware that data was being collected for research purposes, and all user data was kept anonymous.

5.2.1 Study Design

Participants for our survey were recruited through Amazon Mechanical Turk [1]. The only requirement for any participant is that they are 18 years or older of age. The survey consists of 15 questions asking about ownership of varying categories of IoT devices, such as "home assistant" or "fitness tracker". We grouped the questions into three categories - basic demographic information, device ownership information, and participant concerns. The full text of the survey questions can be found in Table 5.1.

We designed the survey to explore and answer the following three study questions:

SQ1) What IoT devices are consumers adopting and using in their daily lives?

SQ2) Which IoT devices are likely to be adopted based on consumer trends?

SQ3) What specific concerns, if any, do consumers have about their IoT devices?

By investigating these questions, we can determine a few key facts about consumer adoption of IoT devices. First, we better understand which, and how many, devices people have already adopted in their daily lives. We gain a snapshot of what types of devices have been popular or not, and understand current interest in the space of IoT as a whole. Second, by understanding trends, we get a

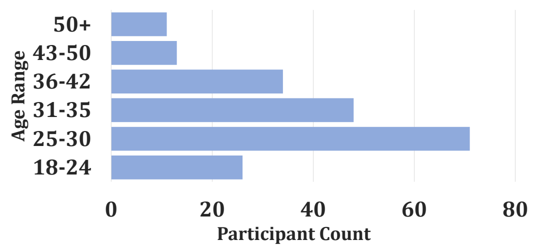
Q#	Full question text
1.	- What is your age range? [18-24], [25-30], [31-35], [36-42], [43-50], [50+]
2.	- On a scale from 1-5 (most comfortable), how would you rate your comfort level with operating technology or electronic devices? [1], [2], [3], [4], [5]
3.	- On a scale from 1-5 (very willing), how would you rate your willingness to engage with new or unfamiliar technology or electronic devices? [1], [2], [3], [4], [5]
4.	- In your own words, briefly describe your understanding of the term “Internet of Things” (IoT) or “Smart Device”. For the purpose of this survey, we do not differentiate between the two terms.
5.	- Do you own any home-assistant devices such as (but not exhaustively): Google Home, Amazon Echo, Amazon Echo Dot, Apple HomePod, etc. If so, please list them below.
6.	- Do you own any wrist devices designed for fitness tracking such as (but not exhaustively): Fitbit, Pebble, Apple Watch, Android Wear watch, Garmin Watch, etc. If so, please list them below.
7.	- Do you own any products designed to intelligently monitor or control the environment around you, such as (but not exhaustively): Nest Thermostat, Nest Smoke Detector, Ecobee Thermostat, Honeywell Thermostat, etc. If so, please list them below.
8.	- Do you own any Internet-connected devices designed to improve home security, such as (but not exhaustively): Bluetooth door locks, door/window sensors, security cameras, etc. If so, please list them below.
9.	- Do you own or use any other Internet-connected IoT devices not listed in the previous questions? If so, please list the device(s) and briefly describe their application. Note: As Smartphones are widely owned, please exclude these device(s) from your answer.
10.	- Are there any IoT device(s) that are on the market, but you do not own, that you wish to own or plan to purchase within the next year? If so, please briefly list the device(s), their application, and why you wish to purchase them.
11.	- Do you have any concerns about the type(s) of data being collected by any of your IoT devices? If so, please briefly describe them.
12.	- Do you have any concerns about the privacy of the data being collected by any of your IoT devices? If so, please briefly describe them.
13.	- Do you have any concerns about how secure any of your IoT devices are? If so, please briefly describe them.
14.	- Do you have any concerns about the reliability of your IoT devices? If so, please briefly describe them.
15.	- Based on your answers to the above four questions, are there any changes you would like to make to any devices that you own to address your concerns? If so, please list the device(s) and describe the change(s) you would make.

Table lines designate question topic groupings.

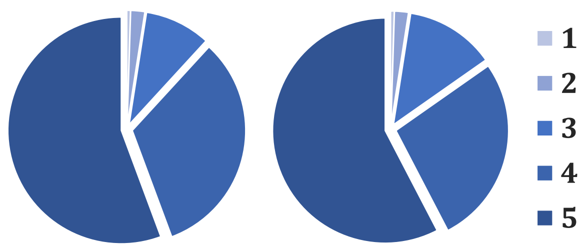
Table 5.1: Full text of the user study questionnaire.

better idea of what devices or categories are most likely to see growth in the future, or which devices will see diminishing interest. Third, by investigating which concerns consumers have about their devices, we can also speculate about the impact these issues will have on consumer purchasing behavior. Furthermore, we can determine which devices are considered sensitive in nature, dealing with highly personal data (e.g. payment information) or scenarios where failure can have broader impacts (e.g. smoke detectors).

5.2.2 Study Demographics



(a) Distribution of participant age range



(b) Participant ratings for comfort (left) and willingness to engage with (right) technology.

Figure 5.1: Distributions of participant demographic information.

Questions 1-3 asked for basic demographics about our participants. Figures 5.1a-5.1b aggregate and display these results. According to Figure 5.1a, we found that approximately half of those who took the survey are under the age of 30, with 75% being under the age of 35. When participants were asked to self-rate their comfort with operating technology on a scale between 1 (uncomfortable) and 5 (most comfortable), we found that 88% of participants replied with a value of either 4 or 5. Similarly, when asked to self-rate their willingness to engage with new technology on a scale from 1 (unwilling) to 5 (very willing), 84% of responses contained a value of either 4 or 5. This indicated that our survey results are taken from a demographic of technologically literate individuals who should be familiar with the device landscape.

Question 4 asked participants to give a short personal definition of the term “Internet of Things” or “Smart Device”. To alleviate confusion, we treat the two terms as synonymous for the survey. Overall, we find that participants are generally familiar with the concept of an IoT device, with only 16% being unfamiliar with or having no understanding of IoT. However, the methods used to express this understanding is widely varied. Table 5.2 summarizes common themes or tropes that were utilized across all definitions. Overall, the most used description is that IoT hardware is connected in some form, with many replies commenting on the Internet connectivity or interconnectivity of these devices. In a similar vein, 21% of replies also commented on the duty of IoT devices to communicate data in some fashion, either to a user or to another device on the network.

Count	Topic Mentioned
72	Internet connection
44	Communicates data to user/other devices
42	Network of interconnected devices
26	Example device(s)
25	Example application scenario(s)
21	Devices make life easier
17	Remote controlled/accessed devices
16	Used in the home
33	Unsure/No understanding

Table 5.2: Summary of common topics in defining “Internet of Things” and “Smart Device”.

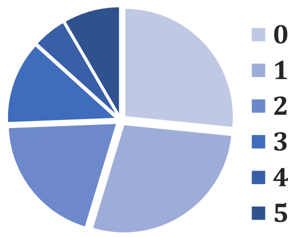
We highlight some of the responses from our participants below.

- “*The Internet of Things is connected devices within a surrounding environment. It is well-used in the home to connect devices.*”
- “*Objects which are not typically internet-enabled or considered ‘computers’ which have been redesigned to take advantage of internet/wi-fi connections in order to interact with apps or other smart devices in order to perform more efficiently*”
- “*All the devices that we currently use at home, but with better technology, internet access and remote control options*”

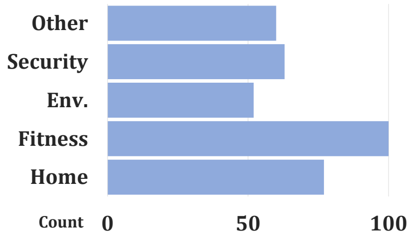
- "*IoT is network of physical devices that are enable to connect with one another and exchange data.*"

5.2.3 Device Ownership

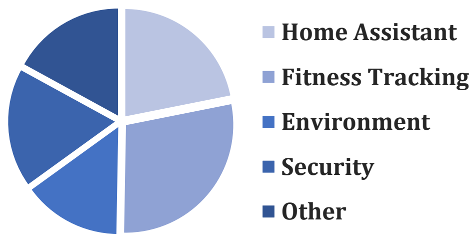
Questions 5-9 investigated general ownership of IoT devices. With these questions, we asked participants to enumerate the devices they own across five categories: *home-assistant*, *fitness tracking*, *environmental monitoring*, *home security*, and *other*. We chose these categories based on a brief investigation of the market. Figure 5.2a illustrates device ownership across all categories. We found that 74% of participants owned an IoT device within one of our specified categories, and by extension owned at least one IoT device. Similarly, we found that 26% of participants owned an IoT device in at least three categories. This demonstrates that adoption of IoT devices is relatively strong and somewhat pervasive.



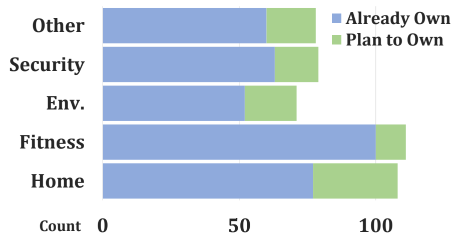
(a) Number of categories participants own devices in



(b) Ownership per category. (out of 203)



(c) Owned devices across categories.



(d) Device ownership adjusted to account for planned purchases. (out of 203)

Figure 5.2: Participant ownership of IoT devices.

Home Assistant	Fitness Tracker	Environment	Security	Other
(30) Amazon Echo	(52) Fitbit*	(18) Nest Thermostat	(44) Security Camera*	(18) Smart TV *
(22) Google Home	(18) Apple Watch	(14) Honeywell Thermostat	(12) Door or window sensor	(12) Tablet*
(12) Echo Dot	(18) Android Wear Watch*	(9) Nest Smoke Detector	(7) Bluetooth Lock*	(11) TV Stream Box*
(126) None	(103) None	(151) None	(140) None	(143) None

Table 5.3: Top 3 owned devices per IoT device category. Ownership count is listed in parenthesis. An asterisk (*) denotes where several similar models were combined

With regards to ownership within the categories themselves, Figure 5.2c illustrates the comparative rate of adoption for each device type. Overall, we found that fitness trackers are the most popular, with a 49% ownership rate (100 owners), while environmental products are the least popular at 25% (52 owners). Home assistants fall in the middle at 38% (77 owners). While this does highlight a degree of variance in product acquisition, it also shows that there is not one category of device completely dominating the market, indicating that a “killer app” may not yet exist.

We also listed the top 3 most popular devices for each category in Table 5.3. Across all categories, we found that the most owned device is some model of the Fitbit fitness tracker, followed by some brand of security camera, and then the Amazon Echo home assistant. This is counter to the trend that indicated more participants owned home assistant devices than security devices, and indicates that home security is an important application for participants to invest in.

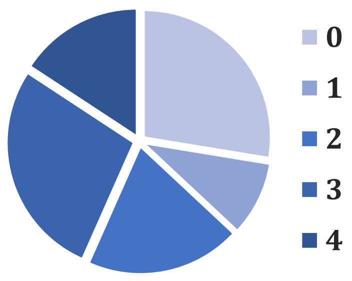
Question 10 asked participants to list devices that they plan to purchase in the next year. Figure 5.2d shows an updated plot of device ownership taking these planned purchases into account. Note, that this plot excludes those who plan on buying devices in categories that they already own a device in. Overall, we saw the biggest jump in home-assistant ownership, bringing the total to 53%, a gain of 15%. Fitness trackers saw the smallest potential growth from 49% to 54%, while all other categories saw potential growth of 8-9%. This indicates to us that home assistant style devices might be popular in concept, but have other factors (e.g.

price, appeal, lack of usage scenarios) preventing mass adoption currently.

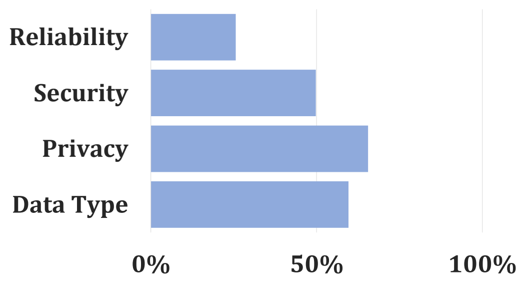
Conclusions Based on our findings, we conclude that fitness trackers have seen the strongest adoption and have established a presence in the consumer IoT device space, while environmental monitoring has grown more slowly. Similarly, we see evidence that consumers are heavily investing in home security with the popularity of security cameras. Over the next year, the data suggests that consumers will be investing in home-assistant devices, which can provide a wide range of functionality to ease everyday life.

5.2.4 Participant Concerns

Questions 11-15 asked participants to describe any concerns they may have about their devices across four categories: the types of data being collected (e.g. heart rate, voice), the privacy of the data being collected (i.e. if they think their data is kept private), the security of their devices, and the reliability of their devices.



(a) Num. of categories participants voiced concerns in.



(b) Concerns of participants vs category of concern.

Figure 5.3: Distribution of participant concern statistics.

Figures 5.3a-5.3b summarize the results for the distribution and quantity of participant concerns. From Figure 5.3a, we find that 66% of users have concerns about the privacy of their data, with worry for the types of data being collected slightly behind it at 60%. Only 50% of participants voiced issues with security, and 26% voiced problems with the reliability of their devices. We similarly plotted the number of categories each participant had a concern in with figure 5.3b. Overall, we found that 73% of participants remarked about at least one category, while 43% mentioned a problem in 3 or 4 (all) categories. This demonstrates that while adoption of these devices is relatively popular, consumers are still not satisfied with many aspects of them.

We categorized some of the common topics of concern expressed by participants in Tables 5.4-5.6. With regards to data types, of the participants that expressed a concern, most mentioned voice and video data, citing examples of home assistant devices (such as the Amazon Echo) being always on and listening. Other types of data, such as location, biometric, behavioral, and financial were mentioned, but with much lower frequency. We find that this correlates with the projected ownership from Figure 5.2d, where home assistant devices showed the biggest growth in ownership. Similarly, the worries about video corresponds with the fact that security cameras were the most owned device in Table 5.3. In short, device ownership and the number of concerns seemed to be correlated. The exception is that, while fitness tracking devices showed the highest percentage of ownership across all categories, health data collection was not a large concern.

With regards to data privacy, participants seemed to focus on five main issues, the largest of which is the belief that stored data is not appropriately handled. Many participants voiced the belief that companies were not responsible in handling their data, either via weak security (#1) or by blatantly collecting and sharing data with third parties (#2). Furthermore, participants noted that they worried about their own data being used against them in some way (#3), or that far more data than necessary was being collected (#4). Responses noted their frustration with not being able to easily see the extent of the data that their devices collected (#5).

With regards to security, participants voiced a number of issues with their devices, the biggest being their frustration with the fact that their devices do not feel secure, or do not seem to be updated to stay ahead of new exploits (#1). Many expressed the fear of information or identity theft, as well as the repercussions of both (#2). We found that users questioned the always-on nature of these de-

#	Concern (data types)
32	* Voice/video
9	* Biometric & Health
9	* Location
9	* Behavioral Information
7	* Financial Information
7	* Personal Info (name/address)

Table 5.4: Most mentioned participant concern topics on data types.

#	Concern (data privacy)
34	* Breaches of stored data
26	* Sharing data with 3rd party
25	* Data Misuse/malicious use
22	* Collecting more data than needed
17	* No control over stored data

Table 5.5: Most mentioned participant concern topics on data privacy.

#	Concern (security)
24	* Devices are insecure, no or infrequent updates
20	* Hacked device $\rightarrow$ identity/info theft
18	* Remote access/access to other devices
13	* Device always on, easy hacking target
11	* Servers/transmission insecure
6	* Default passwords unchanged

Table 5.6: Most mentioned participant concern topics on IoT security.

#	Concern (reliability)
15	* Slow responsiveness/poor or buggy software
12	* Poor hardware, device is quickly obsolete
10	* Doesn't function without power or Internet
7	* Malfunctions cause other problems
5	* Needs repairing, accumulating hidden costs

Table 5.7: Most mentioned participant concern topics on IoT reliability.

vices. This makes them an easy target that's hard to diagnose if hacked (#4), and a convenient entry point into a user's personal network should a hack be successful (#3). Fewer participants worried about the actual transmission of data between devices and the company servers, while even fewer felt that the default security levels of these devices were low, indicating that default passwords were difficult or frustrating to change (#6).

With regards to device reliability, participants had fewer concerns overall. Performance was the biggest concern, with mentions of these devices being slow, or the software being low quality and buggy (#1). Participants also voiced frustration with "planned obsolescence" of hardware, citing low shelf lives of newer gadgets as an issue (#2). Fewer noted that these devices simply cannot function without power or Internet, where an outage would be crippling to functionality (#3).

Less troubling is the thought of a malfunctioning device directly resulting in other problems (#4) or cheap devices having hidden costs, such as requiring frequent repairing or subscription services for proprietary software(#5).

Conclusions We found that, overall 73% of participants concerned with some aspect of their IoT devices. Concerns about the methods in which personal information is handled outweighs how secure the devices are, but not by much. The reliability of these devices tends to be of the least importance. However, several participants voiced their concern about "planned obsolescence" - devices that are not built to last, or are released and then never maintained. We note that the apparent lack of participant issue with malfunctioning devices suggests that, in general, users are either not aware of the broader impacts a misbehaving device could have on other devices.

5.2.5 Study Conclusions

Based on the responses to the survey, we have uncovered the following answers to our research questions.

SQ1 *What IoT devices are consumers adopting and using in their daily lives?* We found that users currently have the most interest in fitness tracking devices, while the least interest is in environmental monitoring devices. The most adopted devices on the market are Fitbit, security cameras, and Amazon Echo.

SQ2 *Which IoT devices are likely to be adopted based on consumer trends?* We found that users are currently most interested in adopting home assistant style devices, like Amazon Echo or Google Home, followed by security cameras.

SQ3 *What specific concerns, if any, do consumers have about their IoT devices?* We found that users are most concerned about the privacy of their data, both on device and in the cloud. Users are least concerned about the reliability of these devices.

5.3 Device Studies

This section presents our investigation into two specific IoT devices to observe what patterns are reflected by RNG use. Specifically, we have chosen to investigate Google Home, and a security camera setup based on open-source MotionEye software [19]. We explain our rationale for choosing these devices below.

5.3.1 Device Selection

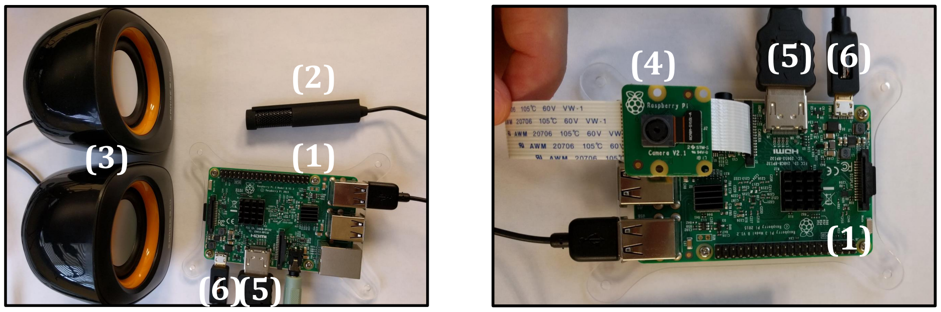
First, we note once more that a majority of users who had a concern about data types collected by IoT devices mentioned either voice and/or video data. Voice data naturally points toward home-assistant style products, where the primary method of interacting with the device is through vocal queries. Similarly, video data points us toward products that are related more toward security, such as security cameras or home monitoring systems. While fitness tracking devices exhibit the greatest adoption among participants, the data they produce (primarily health data) is not considered as sensitive as voice or video.

With regards to adoption, Figure 5.2d suggests home assistant devices will see increased ownership within the next year, matching the popularity of fitness trackers. Furthermore, some of the top concerns with regard to security (#2) and data privacy (#1) surround leakage of private data to third parties, or theft of sensitive information. Home assistants store sensitive information such as voice searches, payment information, purchase histories, etc. Furthermore, they have the ability to act on these data as well (e.g., purchase items online), leading to potential data misuse (data privacy concern #3). Based on these results, we feel that a home assistant IoT device provides a suitable platform for deeper investigation.

With regards to security-related IoT devices, a majority of participants responded that they owned or planned to own some form of security camera. Additionally, Table 5.3 shows that security cameras are the second most owned device across *all* categories. Security monitors and cameras also deal with data of a sensitive nature (e.g., video into private property), necessitating the use of strong encryption to prevent unauthorized access. Because of this, they have been the target of large scale hacks in previous years [95]. This correlates with the privacy concern of data breaches (#1) and security concern of remote access (#3).

5.3.2 Study Setup

In order to facilitate more robust control over the instrumentation of our test devices, we constructed both setups utilizing a Raspberry Pi 3 as the base [30]. Prior research has shown that modifying consumer devices is a difficult task, as the hardware is very isolated and difficult to access [45, 16, 104]. However, Google has provided a hardware and software kit allowing users to assemble a Home Assistant device from commodity parts [40]. Likewise, there are multiple open source security camera software implementations that offer functionality similar to that of closed-source systems [117, 19, 52]. As these software packages run on top of Debian Linux, this allows us to fully monitor random number use at the kernel level.



(a) Google Home (b) Security Camera

Figure 5.4: Hardware setups for the devices used in this study.

Figures 5.4a and 5.4b show the physical setups for our devices, where we label each major hardware component with a number. Both instrumentation setups are built on top of a Raspberry Pi 3B board (1). For the Google Home setup, we use one USB microphone (2) and one pair of stereo speakers (3). For the security camera setup, we use the official Raspberry Pi camera module (4). Both setups are powered via Micro-USB (6) and have an attached HDMI cable (5) for debugging when necessary. We install Raspbian Stretch OS on two separate SD cards and install the necessary software for each setup [30]. Both setups have SSH tunneling enabled to facilitate remote monitoring. Second, as Debian utilizes the Linux PRNG (LPRNG) to produce random numbers, we instrument the kernel source (version 4.9.75-v7) [102] with debugging statements to log when requests for random data are made. Specifically, we log the time since boot, request size, remaining entropy pool size, and process name. Third, in order to mimic the resource constraints of an IoT device, we modify the LPRNG to not accept two of the standard sources that it utilizes: user input (e.g., keyboard and mouse) or disk reads/writes. This mirrors the fact that we expect IoT devices to sit in the background, as well as have limited storage and modes for interaction. Finally, we modify the speed of the Raspberry Pi to match the clock speed of Google Home hardware, which is a Marvell 88DE3006 Armada 1500 Mini Plus dual core processor with 1.3GHz speed [49].

For each device, we empirically determined the scope of distinct activities they can undergo during regular use. In this case, the Home Assistant is our simplest device with one potential activity - listening to and responding to voice queries. The security camera serves as our richer device, performing four main classes of activities: 1) modifying camera settings by utilizing the web interface; 2) triggering the motion detection threshold; 3) taking a still image photograph; 4) recording a video. Given each device and category of event, we perform the necessary steps to trigger the event and record the resulting RNG activity. In total, we perform 100 trials to gather data for each distinct event.

5.3.3 Event Analysis

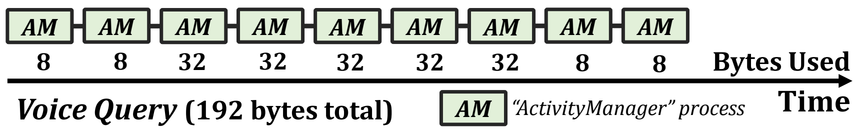
We present our analysis of the events for each device below. To visualize how the sequence of calls fit together for each event, we utilize a block diagram timeline. Each individual box represents one logged call to request random data. For each potential event we transform the raw log data into its corresponding diagram. This process is shown in Figures 5.5a-5.5b. Note that for the source column in the raw logs there are three possible options: Kernel RNG (KRNG), the non-blocking RNG `/dev/urandom` (URNG) and the blocking RNG `/dev/random` (BRNG).

5.3.3.1 Google Home Event

In the Google Home setup, we have exactly one event of interest - performing a voice query to the device. Figure 5.5a shows the raw log, while Figure 5.5b shows the corresponding visual block diagram. In total, there are 9 calls to the non-blocking RNG (URNG), which amounts to 192 bytes of randomness consumed per query. Each individual call in the event is performed by the process `ActivityManager`, which is likely related to the similarly-named `Activity Manager` class found in Android OS.

```
[Timestamp] Source: Req. Size, Process
[2674.864953] URNG:  8, ActivityManager
[2674.864965] URNG:  8, ActivityManager
[2674.888127] URNG: 32, ActivityManager
[2674.888148] URNG: 32, ActivityManager
[2674.888159] URNG: 32, ActivityManager
[2674.965328] URNG: 32, ActivityManager
[2675.000360] URNG: 32, ActivityManager
[2675.024043] URNG:  8, ActivityManager
[2675.024055] URNG:  8, ActivityManager
```

(a) Sample Google Home voice query log.


AM AM AM AM AM AM AM AM AM
8 8 32 32 32 32 32 8 8
Bytes Used
Voice Query (192 bytes total)
AM "ActivityManager" process
Time

(b) Corresponding Google Home voice query execution diagram.

Figure 5.5: (a) Sample kernel log for a Google Home voice query. Data from left to right: Time since boot(s), request source (KRNG, URNG, BRNG), request size (bytes), and requesting process name. (b) Visual execution timeline of a Google Home voice query. The entropy cost of each call (in bytes) is shown beneath the box.

5.3.3.2 Camera Events

Compared to the single event found in Google Home, the security camera setup provides four main categories of events: changing camera settings, motion detection, photo capture, and video recording. Some of these categories have settings which, when changed, will alter the behavior of the RNG. Common to all events is the `relayevent.sh` script (abbreviated as `relay` from here on). Figure 5.6 shows

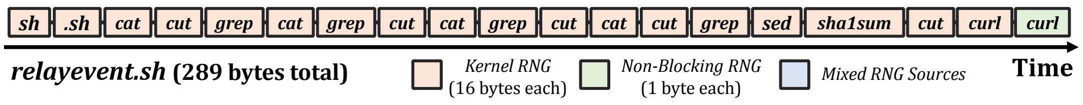

sh
.sh
cat
cut
grep
cat
grep
cut
cat
grep
cut
cat
cut
grep
sed
sha1sum
cut
curl
curl
relayevent.sh (289 bytes total)
Kernel RNG (16 bytes each)
Non-Blocking RNG (1 byte each)
Mixed RNG Sources
Time

Figure 5.6: Visual execution timeline of the `relayevent.sh` script. Each block refers to one process (or process group) making a request for random data. Kernel RNG and Non-Blocking RNG refer to calling `get_random_bytes` and reading `/dev/urandom`, respectively. Colors also apply to Figures 5.7-5.11.

the block diagram for `relay`. In total, `relay` requests 289 bytes across 19 calls each time it is invoked. Note that the coloring legend for Figure 5.6 applies to Figures 5.7-5.11 as well. We now briefly describe the behavior of each camera event.

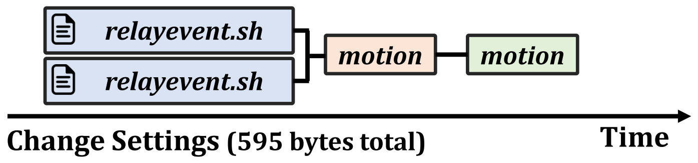


Figure 5.7: Security Camera settings change visual execution diagram.

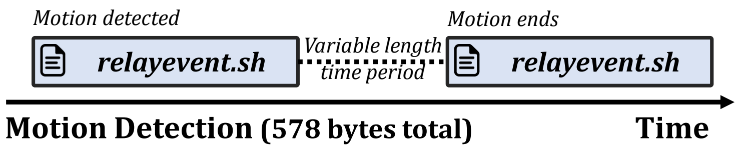

Motion detected
relayevent.sh
Variable length time period
Motion ends
relayevent.sh
Motion Detection (578 bytes total)
Time

Figure 5.8: Security Camera motion detection visual execution diagram.

Settings and Motion Modifying any camera settings via the web GUI (Figure 5.7) results in two simultaneous calls to `relay`, followed by two calls to the `motion` process. Note that the `motion` process does not specifically handle motion detection, but instead acts as the underlying daemon for the MotionEye software. A motion detection event (Figure 5.8) results in one call when the camera's motion tracking threshold is exceeded for a certain number of frames, followed by a second call once there have been a certain number of still frames. Both values are static and can be specified in the GUI settings menu. The camera will attempt to actively highlight the object of interest during the motion period.

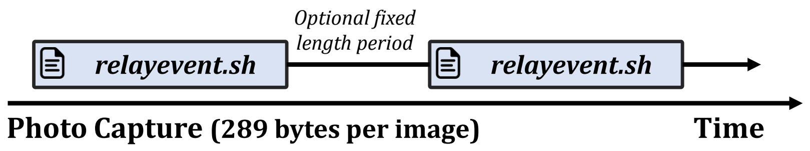

Optional fixed length period
relayevent.sh
relayevent.sh
Photo Capture (289 bytes per image)
Time

Figure 5.9: Security Camera still image visual execution diagram.

Pictures There are several modes of operation for taking a picture (Figure 5.9). First, manually taking a single photo is enabled by the web GUI. Clicking the photo button on a given camera window will result in one call to `relay`. This is similar to when the photo is triggered by motion detection - a single photo will be taken at the start of the motion period which results in one call to `relay`. Photos can also be set to be taken regularly on an interval. This results in one call to `relay` every time the fixed-length waiting period elapses. In general, one photo results in one call to `relay` no matter what the capture scheme is.

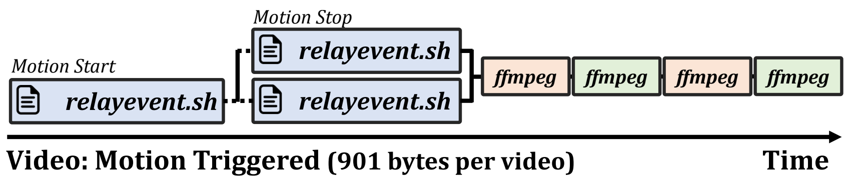


Figure 5.10: Security Camera motion triggered video visual execution diagram.

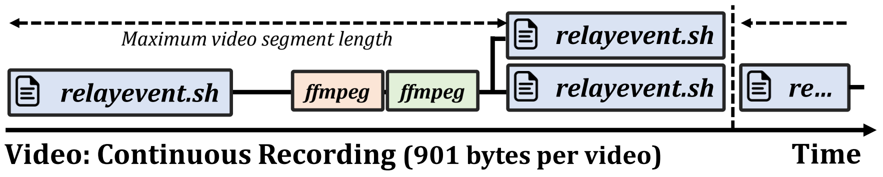
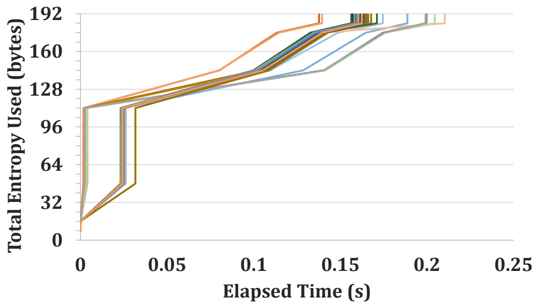

Maximum video segment length
relayevent.sh
relayevent.sh
ffmpeg
ffmpeg
relayevent.sh
re...
Video: Continuous Recording (901 bytes per video)
Time

Figure 5.11: Security Camera continuous video execution diagram.

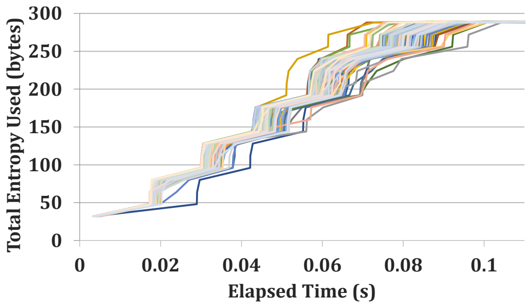
Video There are two modes of operation for recording a video - motion triggered and continuous. When video recording is motion triggered (Figure 5.10), we see an initial call to `relay` when motion is detected, followed by two simultaneous calls to `relay` triggering when motion detection ceases. This is followed by four calls to `ffmpeg`, which deal with encoding the video. When video recording is continuous (Figure 5.11), the following behavior occurs on loop. First, there is a single call to `relay`. A number of seconds after this first call, there are two calls to `ffmpeg`. Finally, a fixed-length period of time after the first call, there are two simultaneous calls to `relay`. The time between the first set of `relay` calls and second set is a fixed time interval and can be set in the GUI. Immediately after the final two calls complete, the loop begins again with the single call to `relay`.

5.3.4 Event Timing

Figures 5.12a and 5.12b plot the total amount of entropy used over the course of a single RNG event against the timing of each call in the event. For simplicity, we look at the voice query event in Google Home, and single picture event in the security camera (which amounts to one call to `relay`). Both figures plot 100 individual traces, where the first call begins at time $t = 0$.



(a) Google Home query



(b) `relayevent.sh` script

Figure 5.12: Entropy use over time for different RNG events.

We make some observations about the variation in total execution length. First, from a total time perspective, a Google Home query occurs over an average of 0.16s, with a range of 0.072s. Similarly, one call of `relay` takes on average 0.08s, with a range of 0.048s. This indicates that, temporally, RNG events are *densely clustered*. With respect to individual call variance, the relative standard deviation for Google Home is 7.7%, while for the camera it is 6.6%. If we consider the variation in timing between calls, there is less consistency, with relative standard deviations ranging from 8% to 312%. However, the much higher variance values are a byproduct of the underlying data being close to zero ($< 10^{-5}$ in some cases). If we instead consider the normal standard deviation, we find on average it is 0.0036s for Google Home and 0.0016s for the security camera. Overall, this indicates while there may be larger degrees of inter-call variance, the overall variance of an entire execution trace is relatively steady. This is best visualized by Figure 5.12b, where the individual traces appear to be very tightly tangled together.

5.3.5 Study Summary

Based on our investigation above, we highlight the following conclusions with respect to RNG access behavior:

- Events that trigger RNG activity are highly structured and consistent between executions. An event consists of groups of different sequences of calls to the RNG.
- The space of potential events in Internet of Things devices is likely to be restricted to a specific, small set, corresponding to different behaviors for that device. This is a byproduct of the generally single-purpose nature of these devices.
- The timing between calls in an RNG event will show some variance, but the overall execution time will be relatively consistent. Furthermore, calls tend to be tightly clustered.

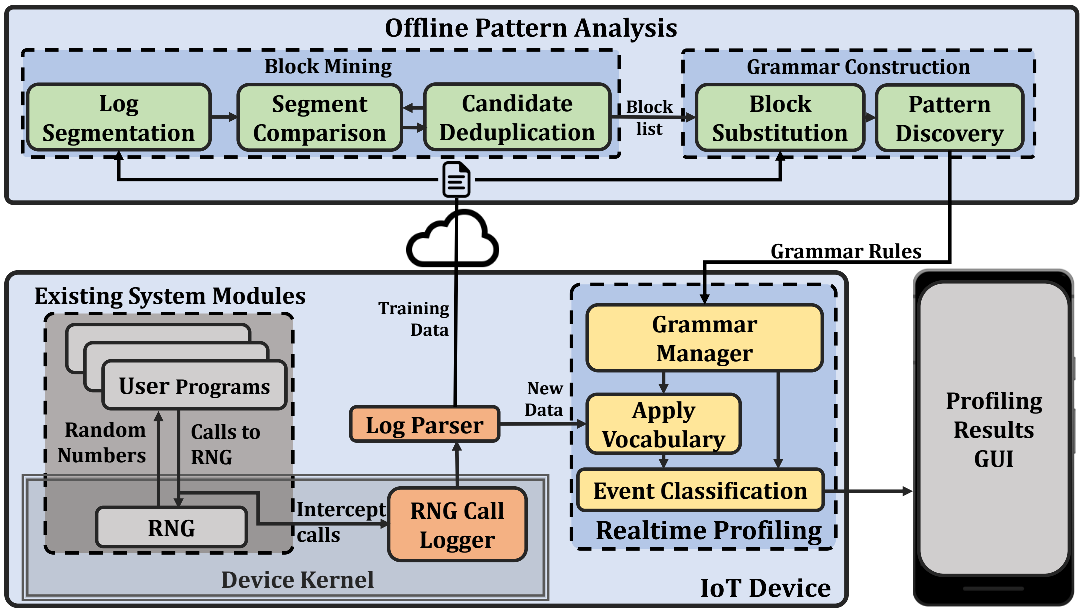


Figure 5.13: The RightNoise architecture. The bottom box represents all modules present on the IoT device, while the top box represents all offline processing steps. The results of profiling are displayed through an external device (e.g. Smartphone).

5.4 RightNoise Architecture

Based on our findings from Section 5.3, we present the architecture of our RNG profiling system called *RightNoise*. At a high level, RightNoise has three main stages - logging, pattern analysis, and live profiling. The pattern analysis stage is performed offline, where aggregated RNG logs are processed to extract meaningful structure in the form of a context free grammar (CFG). Once a grammar is obtained, the IoT device can use it to further characterize its own RNG activity. To facilitate data collection, we modify the kernel with hooks to intercept and log whenever a process requests random data. The whole architecture of RightNoise is shown in Figure 5.13. We discuss the overall flow of data in the system through the rest of this section.

5.4.1 Logging

The first step in the RightNoise pipeline is to extract raw activity data from the RNG. On a Linux system, this is done by instrumenting the `random.c` file in three locations, each corresponding to a different interface that a process can request entropy from (i.e. kernel, `/dev/urandom`, and `/dev/random`). Implementation of this logging hook will need to be done on a per-device basis, depending on what OS the device is running. We record each instance of an RNG access to a file in `/var/log`, logging the time since boot, size of the request (in bytes), targeted interface source, and process name. The log file acts as a buffer for the two different data processing loops.

To minimize the amount of potentially identifying information sent for external processing, the log parser module is also responsible for sanitizing the data. Specifically, it replaces all process names and RNG interfaces with generic strings, where each name is uniquely associated with exactly one string. Once the device being profiled receives the generated grammar back from external analysis, it can perform a reverse substitution to restore the appropriate context.

5.4.2 Offline Pattern Analysis

Pattern analysis of activity logs is done offline before the target IoT device can perform any profiling. We offload the processing of these logs to an external device to remove the impact of resource-intensive algorithms on the device's normal operation. The needed computation for this step can come from many sources. If a user would like to integrate RightNoise according to their own wants, a local edge-computing device, such as an IoT hub (running software such as Mozilla IoT Gateway or openHAB [79, 29, 7]), or a desktop computer on the local network can be used. If the user does not have access to an edge computing device, services in the cloud may be used instead. These computing resources could be provided by the manufacturer of the device as part of their service agreement, or by a third party company offering a device management service.

Our strategy for pattern analysis is derived from our findings in Section 5.3. At a high level, this module takes in raw access logs to the RNG, and outputs the behavior of the RNG in terms of a context-free grammar. We choose a CFG as our representation as they are well researched tools in the field of language modeling [53, 13, 92], and are generally very quick to query once constructed. An

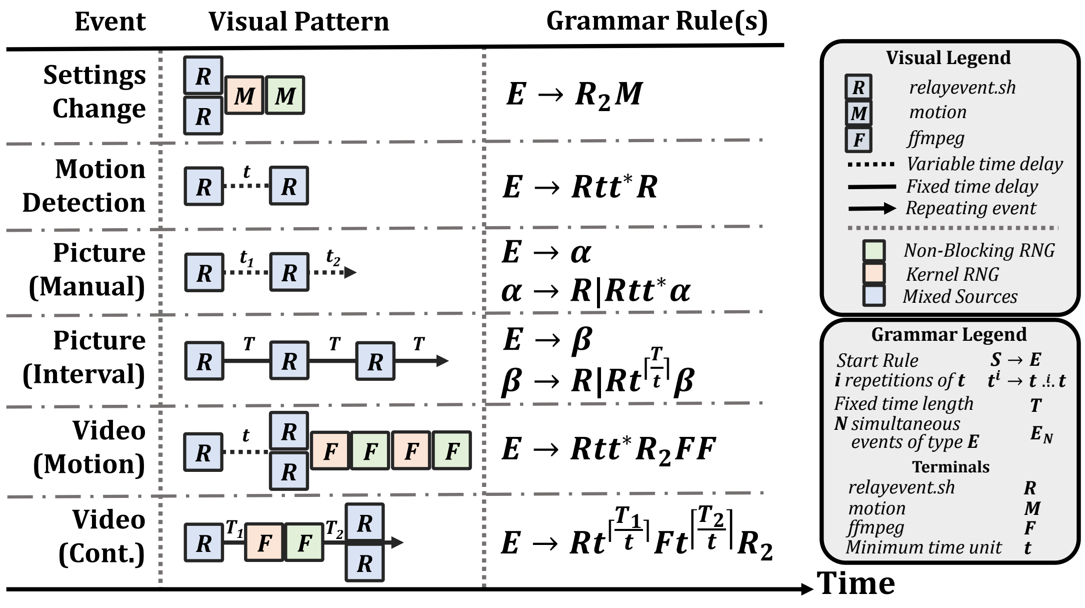


Figure 5.14: Relationship between device activities (left), visual call patterns (center), and grammar rules (right) for the security camera setup. Each row represents one possible activity and the associated RNG access behavior. `t` represents the minimum significant time unit, which is a byproduct of discretization. In total, the language has 4 terminal characters.

intuitive analogy linking RNG activity to natural language is as follows: *individual calls* to the RNG are 'characters', *blocks* of calls (such as scripts) are 'words', and valid 'sentences' are the different *RNG activities* we wish to construct grammar rules for. We cover the algorithms utilized for pattern analysis in greater detail in Section 5.5

We illustrate a concrete example of this natural language concept in Figure 5.14 by hand-crafting a CFG for the security camera setup. Each row in the figure describes one possible device activity (e.g. changing camera settings), the visual representation of the call pattern for that event (middle), and a set of CFG grammar rules that describe that particular event (right). For example, to describe a settings change, the execution diagram shows two simultaneous calls to `relay`, followed by two sequential calls to `motion`. We represent the calls to `relay` with the symbol R_2, and combine the two calls to motion as one symbol, M, leading to the grammar rule $E \rightarrow R_2M$. In total, we collect six meaningfully different activities determined via our device study.

To describe the camera activities, the language has four blocks (words) which wind up being the terminals for the CFG. Three of these blocks correspond to the `relayevent.sh`, pair of calls to `motion`, and pair of calls to `ffmpeg` processes. The remaining block is the minimum significant time delay represented by the terminal `t`. The value of `t` is empirically determined during the pattern analysis algorithm. We note that a fixed time delay `T` (e.g. taking a picture every 10 seconds) can be represented by a corresponding fixed number of instances of the terminal `t`.

5.4.3 Event Detection

Once a set of grammar rules for an IoT device has been created, RightNoise then analyzes the behavior of the RNG while it is running. Intuitively, we compare this problem to understanding a sentence being spoken in real time, where the RNG is the ‘speaker’ and the event detection module is the ‘listener’. As the RNG produces activity, the event detection module determines what language blocks are produced and if the pattern of blocks fits any known grammar rules. If it does, then this segment of activity is labeled accordingly and reported to the GUI.

As requests are made to the RNG, RightNoise aggregates these calls until a significant period of time with no activity occurs. This gap is determined during the offline pattern analysis phase. As the blocks and grammar rules are already known, it can apply the same block substitution algorithm as seen in the grammar construction phase of offline pattern analysis. In addition to substituting language blocks, it also substitutes tokens for time gaps. For data that does not match any known blocks (i.e. *noise* generated by the system from infrequently running processes), we simply label that data as unknown, and ignore when performing pattern matching.

After processing the most recent activity, we attempt to see if it fits into any known grammar rules. If so, we label the relevant blocks of activity accordingly. Because of the nature of online matching, the algorithm may only produce partial matches. For example, the continuous video recording rule in our security camera setup contains two time-gaps. However, the data for only the first, or first and second nontime-gap blocks may have been generated by the device. Therefore, when performing matching against rules, the algorithm also considers the context of the previous k matched blocks, where k is the longest series of nontime-gap blocks across any rule. This may result in reclassification of previous blocks as more context is added. Once a classified block has lapsed out of the recent activity window, the results can be displayed to the GUI. The final reporting of the data can be done on a case-by-case basis. For example, if the device comes with a screen that can be interacted with, the manufacturers may include an app to summarize and display the information right on the device. If the device does not have any meaningful way to display data, the results could be organized and exported to a different device on the network, such as a smartphone, that can interpret the results.

5.5 Offline Pattern Analysis

The offline pattern analysis module is responsible for taking raw RNG access logs and outputting a CFG that describes the RNG's behavior. This module accomplishes that goal in two steps: 1) from raw data, learn what the blocks in the language are; and 2) after knowing what the blocks are, attempt to parse out the most sensible grammar rules. We present our approach to solving this problem over the course of the rest of this section. The top half of Figure 5.13 outlines this process.

5.5.1 Block Mining

Block mining is the process of transforming raw RNG activity (the "letters") and determining what the best candidates for distinct call blocks (the "words") are. This is done as a three stage process: log segmentation, segment comparison, and candidate deduplication.

5.5.1.1 Log Segmentation

In terms of RNG activity, blocks can consist of multiple individual calls that are shared among different events (e.g. the `relayevent.sh` script). As input, it takes an RNG log file containing multiple instances of each different event. The first step is to break up the full RNG log into initial chunks, such that the chunks do not break up any blocks of calls. We do so by computing the distribution of inter-call timings and splitting the log file between calls that are at least 3 standard deviations above the average. We liken this process to taking a word document that has all spaces removed and splitting the word document where any punctuation occurs.

5.5.1.2 Segment Comparison

While the initial splitting algorithm handles obvious breaks between blocks, there are still two problems that need to be addressed. First, it is possible that a chunk which is produced by the log segmentation process could contain multiple valid blocks. For example, the security camera setup has blocks where two instances of `relay` are called simultaneously. To address this, we chose to log one additional piece of data - the `process group id` or PGID, and sort the entries in a single chunk by PGID and then time stamp. This has the effect of keeping calls that are similar together (for example, all the calls in one *relay* script), but also in chronological order.

Once sorting is complete, the segment comparison algorithm can then begin. This algorithm examines all pairs of segments from the segmentation step, looking for shared execution sequences. For example, if one segment contains one execution of a script, and another contains two, we would pull out one execution. Our method for this is seen in Algorithm 1. At a high level, the algorithm iterates through all pairs of segments in the input (lines 3-6), finds the set of common calls between two pairs (line 8), and then computes the Kendall-Tau rank correlation (or permutation distance) metric (lines 10-12) on the common sequences [74]. If the distance is below a certain threshold, we consider the two sequences to be the same and set the common sequence aside (lines 13-14). After all comparisons, the algorithm returns the set of common sequences (line 19).

Algorithm 1 Segment Comparison Algorithm

```
Require: List of segments $S$
Ensure: List of block candidates $C$
 1: $C \leftarrow$ empty list
 2: {Assume each segment in $S$ is pre-sorted by PGID and timestamp}
 3: while $length(S) > 1$ do
 4:   {Compare the first entry in $S$ against all others}
 5:   $segment1 \leftarrow$ = pop front entry off of $S$
 6:   for $segment2$ in $S$ do
 7:     {Find set of common symbols between two segments}
 8:     $CS \leftarrow$ commonSymbols($segment1, segment2$)
 9:     if $length(CS) > 2$ then
10:       $list1 \leftarrow CS \cap segment1$ {Overlapping calls from $segment1$}
11:       $list2 \leftarrow CS \cap segment2$ {Overlapping calls from $segment2$}
12:       $KT \leftarrow KendallTau(list1, list2)$
13:       if $KT/length(CS) < DifferenceThreshold$) then
14:         add $list1$ to $C$ {Two sequences are similar enough}
15:       end if
16:     end if
17:   end for
18: end while
19: return $C$
```

5.5.1.3 Candidate Deduplication

The segment comparison algorithm produces a list of candidate blocks for the RNG grammar. However, because of the nature of the input data, it is very likely that there will be multiple copies of each candidate in the produced data set. Therefore, we perform a deduplication step to remove these copies and reduce the size of the output set. One final problem that needs to be addressed is the possibility that one candidate block could be a subset of another. For example, with the security camera, it could be possible that one candidate represents one call to *relay*, while another candidate represents two simultaneous calls to *relay*. Therefore, we iterate the segment comparison and deduplication steps again until the size of the final candidate list no longer changes. This final candidate list then becomes the block list for the RNG grammar.

5.5.2 Grammar Construction

Once the list of blocks has been determined, the next step is to determine what the valid grammar rules which describe the RNG events are. This is a two step process consisting of block substitution, then pattern discovery.

5.5.2.1 Block Substitution

Block substitution transforms the raw RNG log into a language string by substituting each occurrence of a block with a representative token, thereby reducing the size of the input. In addition to substituting the grammar blocks, we also tokenize time gaps corresponding to the splits determined in step 1 of the block mining process. We note that RightNoise does not currently differentiate between variable and static time gaps with this tokenization process as the pattern discovery module is unable to properly capture their structure.

5.5.2.2 Pattern Discovery

Once block substitution is complete, pattern analysis is performed on the substituted string where it attempts to determine what the most likely valid sentences are in the input. To do this, we formalize the pattern recognition goal as an optimization problem, where the aim is to find the best non-overlapping string covering set. We formalize the problem as follows

Problem Definition: Given an input string S over a fixed alphabet A, with $|A| = k$ and $|S| = N$, we want to find a set of unique substrings $P = \{p_1, p_2, \ldots, p_m\}$, such that:

a) All strings in P are substrings of S

b) S can be reconstructed by concatenating elements of P with replacement

c) The cost of P is minimized according to some cost function $Cost(P)$

We impose a cost function on this problem in order to avoid two trivial solutions. These are the the cases where $P = \{S\}$ and $P = \{A_1, A_2, \ldots, A_k\}$, where

A_i is the i^{th} block in the input alphabet. To build our cost function we formulate two metrics which we refer to as `coverage` and `triviality`. Formal definitions for these metrics can be found in equations 5.1-5.3. Coverage is a measure of how well the input string S can be reconstructed by the elements of P. If every character in S is contained in some element of P, then coverage is maximized. Triviality is a metric that attempts to lower the cost of more 'interesting' solutions. That is to say, solutions where elements of P are used more than once, but elements of P are also longer than one or two blocks. Triviality is a value that we wish to minimize. Our cost function is simply the weighted ratio w between triviality (something we wish to minimize) and coverage (something we wish to maximize). Based on empirical data, we find w to be 2.5.

$$Coverage: \qquad C_P = \sum_{i=1}^{m} |p_i| * n_i \tag{5.1}$$

$$Triviality: \qquad T_P = \sum_{i=1}^{m} (|p_i| + n_i + 1) \tag{5.2}$$

$$Cost: \qquad Cost(P) = \frac{T_P}{(C_P)^w} \tag{5.3}$$

With a cost metric defined, we are able to frame the problem as an optimization problem and attempt to iterate through the solution space. We note that the problem appears to be NP-Hard, so computing an exact solution would be prohibitively expensive. Our approach modifies the technique proposed by B. Stephenson to find an answer to a similar problem - finding the most contributory substring (MCS) of a string [97]. The MCS problem attempts to find a single string p such that the number of occurrences of p times the length of p is maximized within an input string S, or a set of input strings $\{S_1, S_2, \ldots, S_j\}$. This is similar to our problem as it attempts find a substring that best meets a certain metric, but it restricts the size of the solution set to 1. For our problem, we instead want to find a set of substrings $P = \{p_1, \ldots, p_k\}$ such that our cost function is minimized.

To be considered a valid solution, the set P must cover S such that all remaining uncovered substrings in S are unique (i.e. there exists no pattern that can be used more than once). Based on empirical testing, we also modify the MCS scoring algorithm to use the notion of how compressive a substring is instead of simply raw coverage. Compressiveness quantifies how much a given substring will reduce the length of a string, while also factoring in the storage cost of that particular substring. This particular scoring function improved accuracy as it favors selecting shorter substrings that are used multiple times over longer ones, which prevents different language blocks from fusing together in the solution set.

Our solution searching algorithm is described in Algorithm 2. At a high level, it recursively builds up a solution by looking at the most ‘promising’ strings first. For each stage of the recursion, it performs an MCS search (line 7) and orders the results from greatest to least, truncating the list after the top k entries to restrict the recursion width. For each entry in the list (line 8), we create a new set of strings S_{new} by removing all instances of the chosen entry from the input (line 9), and then recursively perform a search on S_{new} (line 10). Once the recursion reaches the base case of having all remaining substrings being unique (line 4), we calculate the cost of the current running solution P (line 5). As the recursion comes back up, it merges all found solutions beneath the current recursion depth, sorts them by cost, and then truncates by the top k (lines 2,13), which is accomplished by using a priority queue of fixed length k. The end result is then transformed into a set of words and grammar rules that can be used by the event detection module on the profiled the IoT device.

5.6 RightNoise Evaluation

In this section, we present our evaluation of RightNoise. We utilize a combination of simulation and real device testing to assess the accuracy and efficacy of the pattern analysis and classification algorithms.

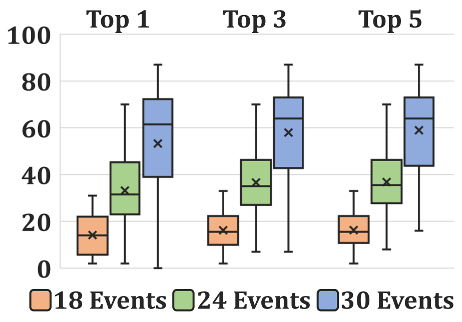
Algorithm 2 Grammar solution searching algorithm

```
Require: List of strings $S$, current running solution $P$
Ensure: Top $k$ lowest cost solutions
 1: function SolutionSearch($S, P$)
 2: Solutions $\leftarrow$ empty PriorityQueue with capacity $k$
 3: {Trivial means all substrings in $S$ are unique}
 4: if $S$ is empty OR $S$ is trivial then
 5:     return (cost(P), P)
 6: end if
 7: Candidates $\leftarrow$ MCS(S,k) {top $k$ MCS results, sorted by score}
 8: for $entry$ in Candidates do
 9:     $S_{new} \leftarrow removeInstances(entry, S)$ {Remove all instances of $entry$ from $S$}
10:     results $\leftarrow SolutionSearch(S_{new}, P \cup entry)$
11:     Solutions $\leftarrow$ Solutions $\cup$ results {Add new solutions into current solutions}
12: end for
13: return Solutions
```

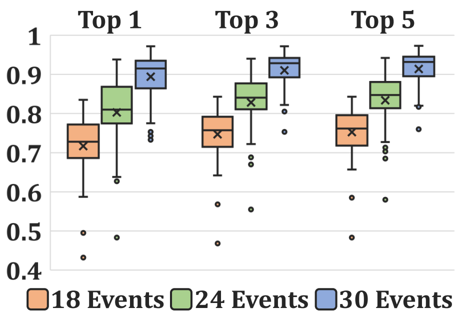
5.6.1 Evaluation Setup

We focus our evaluation on the two main processing loops of RightNoise - the offline pattern analysis, and the real time event classification. In the context of the offline pattern analysis module, we focus our analysis on the accuracy of the algorithms used, as this portion of RightNoise is meant to be performed on devices capable of handling large amounts of computation. For the online classification module, we target both speed and accuracy, as the algorithm is meant to run in real time. For implementation, both the offline pattern analysis and online event detection modules are written in Python, with the former being just under 1000 lines of code and the latter being just over 600 lines of code. For the purpose of executing our simulation study, we use a 40 core server with an Intel Xeon E7-L8867 CPU with 30,720kB cache and 160GB of memory. To get an idea of real-world performance, we implement and test RightNoise on both device setups seen in Section 5.3 - the Google Home and security camera running Motioneye.

The data for our evaluation consists of simulated data, where we randomly construct fictional devices without noise, and real data gathered directly from the devices studied in this paper. The simulated devices are meant to get a broader sense of how well our algorithm performs by simulating other resource constrained devices, such as a smart thermostat, Bluetooth door lock, or smart wristband. In addition, we also gather data from the Google Home and security camera devices used in this paper to show results in the context of the real world. We fix several parameters of our simulated data sets for the sake of comparison to our test devices. The exact parameters are discussed further into the evaluation.



(a) Correctness



(b) Accuracy

Figure 5.15: Top 1, 3, and 5 correctness (a) and accuracy (b) for the pattern recognition module on simulated devices, with input sizes of 18, 24, and 30 events. Note that correctness is measured out of 100, and accuracy is measured in percent.

5.6.2 Pattern Analysis

We focus our evaluation of the offline pattern analysis algorithm on the accuracy of the pattern discovery module. We fix several parameters when creating devices for our simulated data set. Specifically, we set the number of language blocks in a grammar to 4, and the number of events to 6, matching that of the security camera setup. We vary the length of the training data by randomly generating event sequences that are 18, 24, and 30 events long, respectively corresponding to an average of 3, 4, and 5 instances of each event type. We allow the length of a particular event to vary between 2 and 8 blocks. In total, we create 50 simulated devices, and generate 100 example traces for each. For the solution searching algorithm, we fix the recursion width to 3 (i.e. we consider the top 3 best substrings returned by the MCS algorithm at each recursion step). For results, we present how many times the exact correct answer is returned (labeled *correct*), as well as

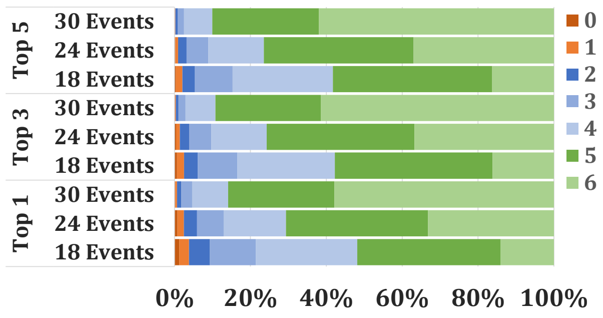


Figure 5.16: Distribution of the most accurate grammars returned across all runs and input sizes. Each color value represents the number of runs with a given amount of correct grammar rules. 6 correct rules indicates the algorithm was able to fully recover the grammar.

the overall accuracy of the best answer in the solution set (labeled *accuracy*). We do this for the top-1, top-3, and top-5 answers returned by the algorithm.

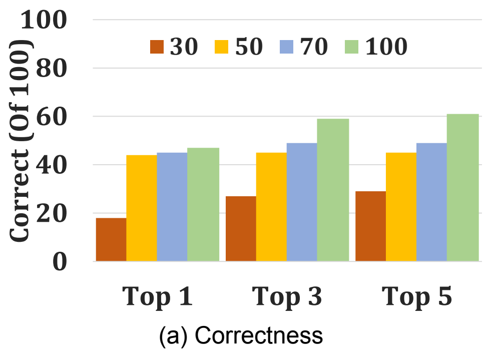
Simulated Data: Figures 5.15a-5.15b summarize the results of the grammar construction algorithm on simulated devices. Given an input length of 30 events (an average of 5 instances of each event per input), our search algorithm is able to find the correct grammar representation an average of 53% of the time if we consider only the top 1 answer returned. This improves to just under 59% when we allow the top 5. According to the median, over 50% of the time, the correct grammar is found 61% of the time, improving to 64% of the time if we allow the top 5. While these numbers may initially seem low, this is a very black and white view of accuracy. If we consider overall accuracy where an answer can be partially correct, we find that, on average, the best returned answer is 89.5% accurate, jumping up to 91.5% accurate if we consider the top 5. This indicates to us that our algorithm is able to find all, or all but one of the events in a grammar a vast majority of the time.

This claim is bolstered in Figure 5.16, where we show a stacked histogram of the solution accuracy across all runs. In total, we find that 88% of the time the the, top solution is expected to have at most one language block incorrect. With smaller input sizes, we see that the accuracy suffers slightly, but the correctness is much lower. Thus, we see that input length has a significant impact on recognition accuracy, but longer input requires more computational time to process.

We investigate the causes of incorrect solutions by comparing the known solution (the *oracle*) to solutions that were returned in the top 5. Overall, we find three interesting patterns with regard to irregularities. First, in some instances our searching algorithm actually does find a lower cost solution than the correct solution. This is usually caused by the algorithm finding a slightly similar or lower coverage solution that greatly reduces the triviality (e.g. by fusing two language blocks together). However, in general, the oracle solution is the lowest cost solution if it is found. Second, we find that the algorithm tends toward overestimating the size of the solution by one or two blocks. In many instances, this is caused by the splitting of one language block into two separate blocks. This effect is likely derived from the fact that we chose to use compressiveness instead of coverage as the scoring function during the MCS search, which tends to favor slightly smaller blocks. However, using the latter metric often resulted in the algorithm underestimating the size of the solution set, which greatly impacted accuracy. Finally, there are the instances where the algorithm simply does not return a lower cost solution than the correct solution, indicating that the recursive search never even considered the option. While this can be addressed by simply increasing the recursion width, this also greatly increases the computation time, a tradeoff which should be carefully considered by the provider of computation.

Device Data: We now evaluate the grammar construction module in terms of two real devices - the Google Home and security camera. We first briefly summarize the results of running the grammar construction algorithm on the Google Home. As the number of grammar rules is limited to two (one rule representing device initialization, and one rule representing a device query), this device prevents any variance in usage patterns other than number of queries per log. Nevertheless, no matter how large the input log grows, our algorithm was always able to recover the correct grammar rules.

We test the security camera setup in a similar fashion to how we tested our simulated device. We collect 100 traces of a particular event length and take the top 1, top 3, and top 5 accuracy of the lowest cost solutions for each. As we are only testing one device in this instance, we also vary the length of the input logs between 30 and 100 events. We summarize our findings in Figures 5.17a-5.17b.



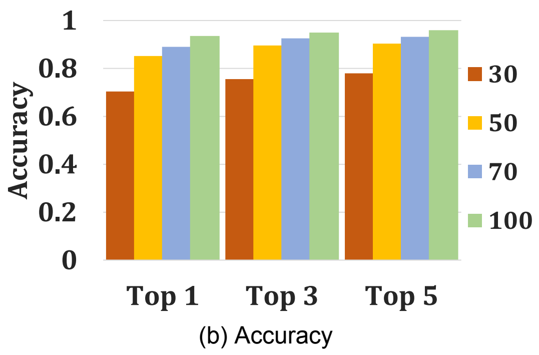


Figure 5.17: Top 1, 3, and 5 correctness (a) and accuracy (b) for the pattern recognition module with data from the security camera setup. Note that correctness is measured out of 100, and accuracy is measured in percent. Different colors correspond to different input lengths.

Overall, as the average number of camera events per log goes up, the accuracy of the algorithm goes up as well, mirroring our findings in the simulated devices. With 30 total events per log (average 5 events), the accuracy is below the average at 78%, even for top 5. However this greatly improves when there are 50 total events per log (average 8.3 events), bringing the accuracy above 80% even for the top 1. As we increase the size of the input logs, the correctness and accuracy of the algorithm continue to improve to above 50/100 for correctness and over 90% for accuracy.

We note that reaching this level of accuracy required more data than expected when compared to the simulated devices, which had over 90% accuracy with an average of 5 of each event type. This is likely due to the structure of the events in the camera themselves. There is a high degree of similarity between the 'motion detection' and 'picture' events. Specifically, motion detection is structured as two copies of the language blocks Rt, while a picture is only one instance of the block Rt. This overlap causes confusion in the pattern discovery algorithm when multiple instances of the language block Rt show up in sequence.

5.6.3 Event Classification

To evaluate the event classification module, we instrument both the Google Home and security camera setups with the RightNoise real time profiling module. To

	SC	MD	P(M)	P(I)	V(MD)	V(C)
SC	1	0	0	0	0	0
MD	0	1	0	0	0	0
P(M)	0	0.27	0.52	0.21	0	0
P(I)	0	0.32	0.21	0.47	0	0
V(MD)	0	0	0	0	1	0
V(C)	0	0	0	0	0	1

Figure 5.18: Confusion matrix for each event in the security camera setup. From top to bottom, the labels represent the 'settings change', 'motion detection', 'picture (manual)', 'picture (interval)', 'video (motion detected)', and 'video (continuous)' events, respectively.

control for any solution noise from the offline pattern analysis module, we encode the grammar discussed in Section 5.4 by hand for each device. For Google Home, the grammar consists of two events (initialization and query). For the security camera, there are six events - settings change, motion detection, manual picture, interval picture, motion triggered video, and continuous video. We proceed to use both devices to generate realistic usage scenarios, producing mixed sequences of 100 events on the Google Home and 400 events on the security camera.

We first discuss the results for the Google Home. The grammar for this device consists of two events - device initialization, and voice query. Both of these events have a distinct structure, represented by the strings `QI,t` and `Q,t` respectively. Because of this fact, recognition accuracy is 100% for all instances of both events. With regards to individual event recognition speed, the module is able to correctly identify each event in just under 0.1 seconds. This duration is measured from the point that the last call in the event is made to the point where the event is successfully labeled by the event detection module. Overall, this indicates that even in devices that may have a reduced activity space, such as a weather monitor, event detection can still achieve high accuracy.

We now discuss the results in the context of the security camera. Again, the grammar for this device consists of six events, with string representations of their structure as follows: settings change (`RR,M,t`), motion detection (`R,t,R,t`), manual picture (`R,t`), interval picture (`R,T`), motion triggered video (`R,t,RR,F,t`), and continuous video (`R,T,F,T,RR`). First, with regards to recognition speed, we find that each event is able to be classified on average in just over 0.1 seconds, demonstrating that the recognition module is very quick. A response time this low facilitates effectively real time communication of a device's behavior to the user. With regards to accuracy, Figure 5.18 shows the confusion matrix for the events generated on the camera. Overall, we see that with the events that have distinct structures - in this case, changing camera settings, motion triggered video, and continuous video - RightNoise is able to achieve perfect identification accuracy. For example, even though the two modes of video capture share the same set of underlying language blocks, there is enough nuance in their ordering that RightNoise is able to easily differentiate. This indicates that when the structure of each activity in a device is diverse enough, RightNoise is able to quickly and easily give meaningful results.

However, with events that are more similar in structure - in this case, motion detection, manual picture taking, and interval picture taking - RightNoise shows much more confusion between the three. For example, we find that all instances where exactly two pictures are taken get classified as motion detection, due to the fact that the number of language blocks in a motion detection event is fixed. Understandably, there is even more confusion between the two modes of picture capture as their structure differs only by the type of time delay. This stems from the fact that RightNoise does not accurately discern the difference between two types of time gaps in our pattern discovery module. However, addressing this problem is left for future work.

5.6.4 Evaluation Summary

In summary, we highlight the following conclusions about RightNoise:

- The offline pattern analysis module is able to achieve high reconstruction accuracy - over 90% when the input data contains on average 5 instances of each distinct activity.

- The real time profiling module is also able to achieve high accuracy, but only in instances where structure between events is sufficiently different.

- The profiling module is able to classify different events, on average, in 0.1 seconds after event completion.

5.7 Future Applications of RightNoise

With the construction and demonstration of the capabilities of RightNoise, we take this section to discuss possible directions for future research in this area, both to improve upon the current architecture or applications of the tool itself.

Detection of Botnet Infection In August 2016, devices across the Internet of Things were infected by a botnet called Mirai, spreading to an estimated 600,000 devices [4]. This virtual army was used to carry out several DDoS attacks over the next few months, targeting several key web infrastructure points and rendering access to major websites impossible for several hours at a time [65]. While Mirai is by no means the first infection to spread through the space of IoT devices[10], the sheer size of the DDoS attacks triggered research on understanding how it spread and how to best address these problems. However, while preventing the spread of malicious code in the future is an important goal, being able to quickly detect when a device is infected is crucial for helping stop the spread of newer Malware that relies on unknown exploits.

A common behavior that Malware uses to spread is a scan and brute force loop. First, an infected device will randomly randomly scan the IP space on the local network to try and find a new target that will respond to it. Once it gets a response, it will use a myriad of methods to get in, including brute forcing default passwords, checking for low quality passwords, checking for known manufacturer vulnerabilities, and spending a large amount of computation to try and gain access to the targeted device. In doing so, however, these behaviors have the potential to generate a lot of unique activity on the infected device, potentially in the RNG.

If an infected device comes with RightNoise integrated, it would be relatively easy to notice a change in behavior given that device had been profiled during a known good state (e.g., immediately after device setup is completed). The extra RNG activity generated by the malicious programs would be categorized as “unclassified”. If enough activity is generated, RightNoise could raise an alert that there is an influx of new activity on the device, giving the user a message to ensure that their device is functioning correctly.

User Activity Chronicling Certain classes of IoT devices depend on user interaction as part of their utility. For example, a home assistant device requires the user to put forth a voice query to receive desired information. Similarly, a security camera setup has several actions, where a user can view a remote feed, take a picture, or record a video, among others. However, depending on the device and the implementation, logs of these activities may not be available for device owners to view or access, which is a barrier to understanding the behavior of a device. As we show in this paper, RightNoise is able to implicitly draw parallels between calls to the RNG and device activities by building a model to associate these behaviors.

However, as this link is only implicit, the resulting data displayed lacks any relevant labels that describe the detected activities in the device. This results in a knowledge gap between the user and device for non-experts. However, there are several ways this gap can be addressed. First, RightNoise could offer a supervised labeling mode in the GUI, wherein the user triggers an event in the device, RightNoise returns the resulting relevant data, and the user can then apply a label to the data for future use. This would require extra manual effort from the owner of the device. An alternative is that the device manufacturer could ship their product with a pre-labeled set of activities. Similarly, enthusiasts could maintain an open source repository to catalog label sets for different types of devices, along with instructions for users to easily install them.

With activity labeling in place, further information could potentially be gleaned by the user about the state of their device. For instance, if a picture is scheduled to be taken every hour in a security camera setup, but there is no corresponding activity in the RNG log, the user might suspect something has gone wrong. Similarly, if a malicious entity has gained unauthorized access to a device, and there are periods of activity at unusual times (e.g. changing the camera settings while the user is usually asleep), then this could also be considered unusual behavior and alert the user.

5.8 Future Work

The current pattern mining algorithm of RightNoise makes a few key assumptions that allow it to function within the context of an IoT device. First, we expect that the volume of activity will be rather low, to the point that there should be no overlap between any two events a vast majority of the time. While this generally holds true for resource-constrained, single purpose devices, it does not extend to more traditional computer or server environments where the chance for simultaneous activity is much higher. In this case, it is likely that the block mining algorithm for RightNoise will break down and have difficulty pulling apart any events that are intertwined. Therefore, research into more robust pattern mining algorithms will increase the applicable domain of the system.

Second, RightNoise currently has no sufficient way to detect and differentiate between the two types of time gaps - fixed-length and variable-length. This in turn reduces the granularity of the rules that it is able to generate. For example, in the security camera setup, it is difficult for RightNoise to differentiate between two pictures being taken manually, and two pictures being taken on a timer. This shortcoming would need to be solved on two fronts - the pattern mining algorithm would need to perform extra analysis to determine what category each time gap likely falls under, and the profiling algorithm needs to take previous context into account to accurately identify the types of time gaps as they happen.

In a similar vein, RightNoise struggles to differentiate between two events when their associated rules are very close in structure. For example, detecting motion in the security camera always generates exactly two instances of *relay*. However, this is indistinguishable from two pictures being taken manually. Without additional contextual information, this problem may be difficult to address with strictly RNG activity alone.

Finally, the method used to get the RNG activity data currently involves modifying and recompiling the kernel. While the modifications are not overly complex, this puts a high barrier to adoption for devices that have restrictions on if the kernel can be modified or not (e.g. mobile phones). In the future, alternative methods for logging RNG activity will be needed to lower the cost of adoption. Alternatively, a standardized implementation for a kernel module could be agreed upon and integrated into the source, mitigating the need to recompile the kernel for every device.

5.9 Conclusion

Due to the nature of the Internet of Things, devices are often left unattended or unmonitored, often causing unwanted behavior to go unnoticed for longer periods of time. This paper presents the first investigation into characterizing device behavior from RNG activity logs. From this investigation, we build a tool called RightNoise, which utilizes a combination of grammar-based offline pattern analysis and real time event detection to profile and report the behavior of an IoT device with over 90% accuracy. With the ability to differentiate between known and unknown behavior, RightNoise can be leveraged in detecting problems in the future, such as widespread botnet infections.

Chapter 6

Conclusion

This dissertation presents three projects designed to increase the understanding of and enrich the quality of randomness within the context of resource-constrained devices.

First, we present 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, suggesting the viability of a fully optimized solution.

Second, we alleviate the problem of entropy stagnation within the context of resource-constrained devices by proposing the initial designs for a Collaborative and Distributed Entropy Transfer (CADET) protocol. With this protocol, devices that have generated an excess of entropy can indirectly assist those that are entropy deficient. Throughout this paper we have highlighted a number of design choices taken in order to maximize efficiency of the framework, utilizing a testbed of 49 Raspberry Pi 3B devices to gather additional supporting evidence. The groundwork has been laid for future work on this topic, with a number of open questions still remaining for exploration.

Finally, due to the nature of the Internet of Things, devices are often left unattended or unmonitored, causing unwanted behavior to go unnoticed for longer periods of time. We present the first investigation into characterizing device behavior from RNG activity logs. From this investigation, we build a tool called RightNoise, which utilizes a combination of grammar-based offline pattern analysis and real time event detection to profile and report the behavior of an IoT device with over 90% accuracy. With the ability to differentiate between known and unknown behavior, RightNoise can be leveraged in detecting problems in the future, such as widespread botnet infections.

Bibliography

[1] Amazon. Amazon mechanical turk. `https://www.mturk.com/`, 2016.

[2] Syed Obaid Amin, Muhammad Shoaib Siddiqui, Choong Seon Hong, and Jongwon Choe. A novel coding scheme to implement signature based ids in ip based sensor networks. In *Integrated Network Management-Workshops, 2009. IM'09. IFIP/IEEE International Symposium on*, pages 269–274. IEEE, 2009.

[3] Kishore Angrishi. Turning internet of things (iot) into internet of vulnerabilities (iov): Iot botnets. *arXiv preprint arXiv:1702.03681*, 2017.

[4] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *USENIX Security Symposium*, 2017.

[5] Apple. iOS Security. Technical report, Apple Inc., February 2014.

[6] Briana Arrington, LiEsa Barnett, Rahmira Rufus, and Albert Esterline. Behavioral modeling intrusion detection system (bmids) using internet of things (iot) behavior-based anomaly detection via immunity-inspired algorithms. In *Computer Communication and Networks (ICCCN), 2016 25th International Conference on*, pages 1–6. IEEE, 2016.

[7] Home Assistant. Home assistant. `https://homeassistant.io`, 2017.

[8] Elad Barkan, Eli Biham, and Nathan Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. *J. Cryptol.*, 21(3):392–429, March 2008.

[9] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[10] Elisa Bertino and Nayeem Islam. Botnets and internet of things security. *Computer*, 50(2):76–79, 2017.

[11] Gregory J Chaitin. *Information, randomness & incompleteness: papers on algorithmic information theory*, volume 8. World Scientific, 1990.

[12] Corbin Champion. Addi application source. `https://play.google.com/store/apps/details?id=com.addi&hl=en`, 2012.

[13] Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. *AAAI/IAAI*, 2005(598-603):18, 1997.

[14] Jim Cheetham. Onerng - open hardware random number generator. `http://onerng.info`, 2014.

[15] Alexander Chefranov, SeyedMasoud Alavi Abhari, Hooman Alavizadeh, and Maryam Farajzadeh Zanjani. Secure True Random Number Generator in WLAN/LAN. In *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN '13, pages 331–335, New York, NY, USA, 2013. ACM.

[16] I. Clinton, L. Cook, and S. Banik. White paper: A survey of various methods for analyzing the amazon echo. Technical report, School of Mathematics and Computer Science, The Citadel South Carolina, 2016.

[17] Ben Eleventh Consulting. haveged - a simple entropy daemon. `http://www.issihosts.com/haveged/`, 2016.

[18] Henry Corrigan-Gibbs, Wendy Mu, Dan Boneh, and Bryan Ford. Ensuring High-quality Randomness in Cryptographic Key Generation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 685–696, New York, NY, USA, 2013. ACM.

[19] Calin Crisan. Motioneye wiki. `https://github.com/ccrisan/motioneye/wiki`, 2015.

[20] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable. In *Proceedings of NDSS' 14*, San Diego, CA, USA, February 2014. ACM.

[21] Yu Ding, Zhou Peng, and Chao Zhang. Android Low Entropy Demystified. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, June 2014.

[22] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergnaud, and Daniel Wichs. Security Analysis of Pseudo-Random Number Generators with Input: /dev/random is not Robust. *International Association for Cryptologic Research*, pages 1–31, 2013.

[23] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 77(2-3):493–514, 2015.

[24] Donald E Eastlake, Stephen D Crocker, and Jeffrey I Schiller. Randomness requirements for security. Technical report, RFC 1750, Dec, 1994.

[25] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Emprircal of Cryptographic Misuse in Android Applications. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, November 2013.

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

[27] Joseph R Enzminger. Method, apparatus, and program product for distributing random number generation on a gaming network, March 4 2014. US Patent 8,662,994.

[28] Adam Everspaugh, Yan Zhai, Robert Jellinek, Thomas Ristenpart, and Michael Swift. Not-so-random numbers in virtualized linux and the whirlwind rng. In *2014 IEEE Symposium on Security and Privacy*, pages 559–574. IEEE, 2014.

[29] OpenHAB Foundation. Openhab. `https://www.openhab.org`, 2016.

[30] Raspberry Pi Foundation. Raspberry pi resources. `https://raspberrypi.org`, 2014.

[31] Aurélien Francillon and Claude 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*, pages 1–7. IEEE, 2007.

[32] Vincenzo Gaglio, Alessandra De Paola, Marco Ortolani, and Giuseppe 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*, Q2SWinet '10, pages 82–89, New York, NY, USA, 2010. ACM.

[33] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia 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*, SenSys '14, pages 295–309, New York, NY, USA, 2014. ACM.

[34] 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)*, pages 72–81, 2013.

[35] Google. Randommusicplayer application source code. https://github.com/android/platform_development/tree/master/samples/RandomMusicPlayer/src/com/example/android/musicplayer, 2012.

[36] Google. Android developer documentation. https://developer.android.com/guide/index.html, 2014.

[37] Google. Android activity manager profiler shell commands. http://developer.android.com/tools/help/shell.html, 2016.

[38] Google. Android getevent shell command. https://source.android.com/devices/input/getevent.html, 2016.

[39] Google. Android things. https://developer.android.com/things/index.html, 2017.

[40] Google. Google aiy voice kit. https://aiyprojects.withgoogle.com/voice, 2017.

[41] Google. Google weave. https://developers.google.com/weave/, 2017.

[42] Parikshit Gopalan, Raghu Meka, Omer Reingold, Luca Trevisan, and Salil Vadhan. Better pseudorandom generators from milder pseudorandom restrictions. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pages 120–129. IEEE, 2012.

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

[44] Zvi Gutterman and Benny Pinkas. Analysis of the Linux Random Number Generator. *International Association for Cryptologic Research*, pages 1–18, March 2006.

[45] William Haack, Madeleine Severance, Michael Wallace, and Jeremy Wohlwend. Security analysis of the amazon echo. https://courses.csail.mit.edu/6.857/2017/project/8.pdf, 2017.

[46] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220, Bellevue, WA, 2012. USENIX.

[47] Christine Hennebert, Hicham Hossayni, and Cédric Lauradoux. Entropy harvesting from physical sensors. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 149–154. ACM, 2013.

[48] Elike Hodo, Xavier Bellekens, Andrew Hamilton, Pierre-Louis Dubouilh, Ephraim Iorkyase, Christos Tachtatzis, and Robert Atkinson. Threat analysis of iot networks using artificial neural network intrusion detection system. In *Networks, Computers and Communications (ISNCC), 2016 International Symposium on*, pages 1–6. IEEE, 2016.

[49] iFixit. Google home teardown. https://www.ifixit.com/Teardown/Google+Home+Teardown/72684, 2016.

[50] Whitewood Encryption Systems Inc. Quantum entropy-as-a-service. https://www.getnetrandom.com/, 2017.

[51] Intel. Intel rdrand instruction documentation. https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide, 2016.

[52] iSpyConnect. ispy open source video surveillance software. https://www.ispyconnect.com, 2007.

[53] Frederick Jelinek, John D Lafferty, and Robert L Mercer. Basic methods of probabilistic context free grammars. In *Speech Recognition and Understanding*, pages 345–360. Springer, 1992.

[54] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z Morley Mao, Atul Prakash, and Shanghai JiaoTong Unviersity. Contexiot: Towards providing contextual integrity to appified iot platforms. In *Proceedings of The Network and Distributed System Security Symposium*, volume 2017, 2017.

[55] Chen Jun and Chen Chi. Design of complex event-processing ids in internet of things. In *Measuring Technology and Mechatronics Automation (ICMTMA), 2014 Sixth International Conference on*, pages 226–229. IEEE, 2014.

[56] k9mail Team. k9 mail application source code. `https://github.com/k9mail/k-9`, 2015.

[57] David Kaplan, Sagi Kedmi, Roee Hay, and Avi Dayan. Attacking the linux prng on android: Weaknesses in seeding of entropic pools and low boot-time entropy. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, San Diego, CA, August 2014. USENIX Association.

[58] Prabhakaran Kasinathan, Gianfranco Costamagna, Hussein Khaleel, Claudio Pastrone, and Maurizio A Spirito. An ids framework for internet of things empowered by 6lowpan. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1337–1340. ACM, 2013.

[59] Prabhakaran Kasinathan, Claudio Pastrone, Maurizio A Spirito, and Mark Vinkovits. Denial-of-service detection in 6lowpan based internet of things. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*, pages 600–607. IEEE, 2013.

[60] 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, 1999.

[61] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *International Workshop on Selected Areas in Cryptography*, pages 13–33. Springer, 1999.

[62] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. Predictability of android openssl's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 659–668. ACM, 2013.

[63] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. Predictability of Android OpenSSL's Psuedo Random Number Generator. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 659–668. ACM, November 2013.

[64] Dustin Kirkland. Ubuntu pollen. `https://github.com/dustinkirkland/pollen`, 2014.

[65] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[66] Jan Krhovjak, Petr Švenda, and Václav Matyáš. The Sources of Randomness in Mobile Devices. In *Proceedings of the 12th Nordic Workshop on Secure IT Systems*. NordSec, October 2007.

[67] James Kurose. *Computer Networking*. Pearson, sixth edition, 2013.

[68] Patrick Lacharme, Andrea Röck, Vincent Strubel, and Marion Videau. The Linux Pseudorandom Number Generator Revisited. *International Association for Cryptologic Research*, pages 1–23, 2012.

[69] Nicholas D. Lane, Yohan Chon, Lin Zhou, Yongzhe Zhang, Fan Li, Dongwon Kim, Guanzhong Ding, Feng Zhao, and Hojung 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, SenSys '13, pages 7:1–7:14, New York, NY, USA, 2013. ACM.

[70] Anhtuan Le, Jonathan Loo, Yuan Luo, and Aboubaker Lasebae. Specification-based ids for securing rpl from topology attacks. In *Wireless Days (WD), 2011 IFIP*, pages 1–3. IEEE, 2011.

[71] F-Droid Limited. F-droid android application repository. `https://f-droid.org/`, 2015.

[72] George Marsaglia. Diehard battery of tests of randomness, 1995.

[73] Robert McEvoy, James Curran, Paul Cotter, and Colleen Murphy. Fortuna: cryptographically secure pseudo-random number generation in software and hardware. In *Irish Signals and Systems Conference, 2006. IET*, pages 457–462. IET, 2006.

[74] A Ian McLeod. Kendall rank correlation and mann-kendall trend test. *R Package Kendall*, 2005.

[75] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[76] Grace Metri, Weisong Shi, and Monica 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*, HotMobile '15, pages 39–44, New York, NY, USA, 2015. ACM.

[77] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. Randomly Failed! The State of Randomness in Current Java Implementations. In *Proceedings of the 13th International Conference on Topics in Cryptology*, CT-RSA'13, pages 129–144, Berlin, Heidelberg, 2013. Springer-Verlag.

[78] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. Gyrophone: Recognizing Speech from Gyroscope Signals. In *23rd USENIX Security Symposium*

(USENIX Security 14), pages 1053–1067, San Diego, CA, August 2014. USENIX Association.

[79] Mozilla. Things gateway. `http://iot.mozilla.org/gateway/`, 2017.

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

[81] Julien Niset and Louis-Philippe Lamoureux. Random number distribution, December 5 2014. US Patent App. 14/562,308.

[82] NIST. Nist randomness beacon. `http://www.nist.gov/itl/csd/ct/nist_beacon.cfm`, 2012.

[83] Jesus Pacheco and Salim Hariri. Iot security framework for smart cyber infrastructures. In *Foundations and Applications of Self* Systems, IEEE International Workshops on*, pages 242–247. IEEE, 2016.

[84] Jesus Pacheco and Salim Hariri. Anomaly behavior analysis for iot sensors. *Transactions on Emerging Telecommunications Technologies*, 29(4):e3188, 2018.

[85] Brian Pellin. Keepassdroid application source code. `https://github.com/bpellin/keepassdroid`, 2015.

[86] Qualcomm. Trepn profiler. `https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler`, 2015.

[87] Randomness and Ltd. Integrity Services. Random.org true random number service. `http://www.random.org`, 2015.

[88] Shahid Raza, Linus Wallgren, and Thiemo Voigt. Svelte: Real-time intrusion detection in the internet of things. *Ad hoc networks*, 11(8):2661–2674, 2013.

[89] Giuseppe Lo Re, Fabrizio Milazzo, and Marco Ortolani. Secure Random Number Generation in Wireless Sensor Networks. In *Proceedings of the 4th international conference on Security of information and networks*, pages 175–182. ACM, November 2011.

[90] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.

[91] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. *National Institute of Standards and Technology*, 800-22(1a):1–131, April 2010.

[92] Yasubumi Sakakibara, Michael Brown, Richard Hughey, I Saira Mian, Kimmen Sjölander, Rebecca C Underwood, and David Haussler. Stochastic context-free grammers for trna modeling. *Nucleic acids research*, 22(23):5112–5120, 1994.

[93] Bruno Sanguinetti, Anthony Martin, Hugo Zbinden, and Nicolas Gisin. Quantum random number generation on a mobile phone. *arXiv preprint arXiv:1405.0435*, 2014.

[94] Anna Kornfeld Simpson, Franziska Roesner, and Tadayoshi Kohno. Securing vulnerable home iot devices with an in-hub security manager. In *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*, pages 551–556. IEEE, 2017.

[95] Ms. Smith. Peeping into 73,000 unsecured security cameras thanks to default passwords, 2014.

[96] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, sixth edition edition, 2013.

[97] Ben Stephenson. An efficient algorithm for identifying the most contributory substring. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 272–282. Springer, 2007.

[98] Alin Suciu, Daniel Lebu, and Kinga Marton. Unpredictable random number generator based on mobile sensors. In *Intelligent Computer Communica-*

tion and Processing (ICCP), 2011 IEEE International Conference on, pages 445–448. IEEE, 2011.

[99] Berk Sunar, William J. Martin, and Douglas R. Stinson. A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks. *IEEE Trans. Comput.*, 56(1):109–119, January 2007.

[100] Aaron Toponce. True random number service - entropy as a service. `http://hundun.ae7.st/`, 2012.

[101] Linus Torvalds. Linux prng source code. `https://github.com/torvalds/linux/blob/master/drivers/char/random.c`, 2014.

[102] Linus Torvalds. Linux kernel source. `https://github.com/torvalds/linux`, 2015.

[103] ubldit. Truerng - hardware random number generator. `http://ubld.it/products/truerng-hardware-random-number-generator/`, 2013.

[104] Alex Vanderpot. Echohacking wiki. `https://github.com/echohacking/wiki/wiki`, 2017.

[105] Apostol Vassilev and Robert Staples. Entropy as a service: Unlocking cryptography's full potential. *Computer*, 49(9):98–102, 2016.

[106] Jonathan Voris, Nitesh Saxena, and Tzipora Halevi. Accelerometers and Randomness: Perfect Together. In *Proceedings of the Fourth ACM Conference on Wireless Network Security*, WiSec '11, pages 115–126, New York, NY, USA, 2011. ACM.

[107] Thubaut Vuillemin, François Goichon, Cédric Lauradoux, and Guillaume Salagnac. Entropy Transfers in the Linux Random Number Generator. Technical Report 1, Grenoble Research Center, 655 Avenue de l'Europe Montbonnot, September 2012.

[108] John Walker. Hotbits - genuine random numbers. `https://www.fourmilab.ch/hotbits/`, 1996.

[109] Kyle Wallace, Kevin Moran, Ed Novak, Gang Zhou, and Kun Sun. Toward sensor-based random number generation for mobile and iot devices. *IEEE Internet of Things Journal*, 3(6):1189–1201, 2016.

[110] Rolf H Weber. Internet of things–new security and privacy challenges. *Computer law & security review*, 26(1):23–30, 2010.

[111] Wolfram. Random number generation. `http://reference.wolfram.com/language/tutorial/RandomNumberGeneration.html`, 2014.

[112] Matthew Wood and Gary Graunke. Enhancing entropy in pseudo-random number generators using remote sources, March 30 2001. US Patent App. 09/822,548.

[113] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 5. ACM, 2015.

[114] Kai Zhao and Lina Ge. A survey on the internet of things security. In *Computational Intelligence and Security (CIS), 2013 9th International Conference on*, pages 663–667. IEEE, 2013.

[115] Liang Zhou and Han-Chieh Chao. Multimedia traffic security architecture for the internet of things. *IEEE Network*, 25(3), 2011.

[116] Zoff. Aagtl geocachingtool for android. `http://aagtl.work.zoff.cc`, 2012.

[117] Zoneminder. Zoneminder camera software. `https://zoneminder.com`, 2015.