CADET: Investigating a Collaborative and Distributed Entropy Transfer Protocol

Kyle Wallace, Gang Zhou
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185
{kmwall,gzhou}@cs.wm.edu

Kun Sun
Department of Information Sciences and Technology
George Mason University
Fairfax, VA 22030
{ksun3}@gmu.edu

***Abstract*—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.**

***Index Terms*—Random number generation, Internet of Things, Entropy, Collaboration, distributed service.**

I. 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 [1], [2].

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 [3], while a PRNG uses a combination of mathematical operations and initial "seed" data to produce a stream of statistically random values [4].

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 [4] [5]. 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 [6], [7].

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 [8]. Similarly, consumer devices effectively put a TRNG in a black box (e.g., USB stick, Smart Card) to augment on-board implementations [9], [10]. 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) [11], [12], [13], [14], [15], 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 [3], [16]. Multiple patents discussing various mechanisms for distributing entropy have been submitted [17], [18], [19]. Most notably, a centralized entropy service was proposed by the National Institute of Standards and Technology (NIST) [6]. However, to the best of the authors' 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:

1) 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.

2) 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.

3) 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.

II. 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 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.

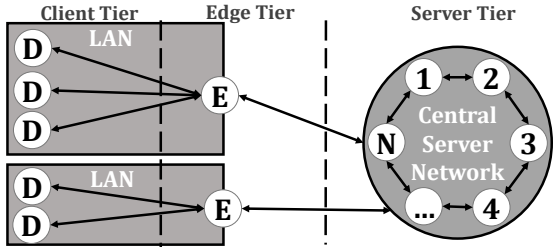


Fig. 1. The CADET device topology. The server network is a collection of 1 to N devices which host entropy data. Edge nodes (E) bridge the gap between the local network (where client devices (D) are) and the server network.

The 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 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? (§III)

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

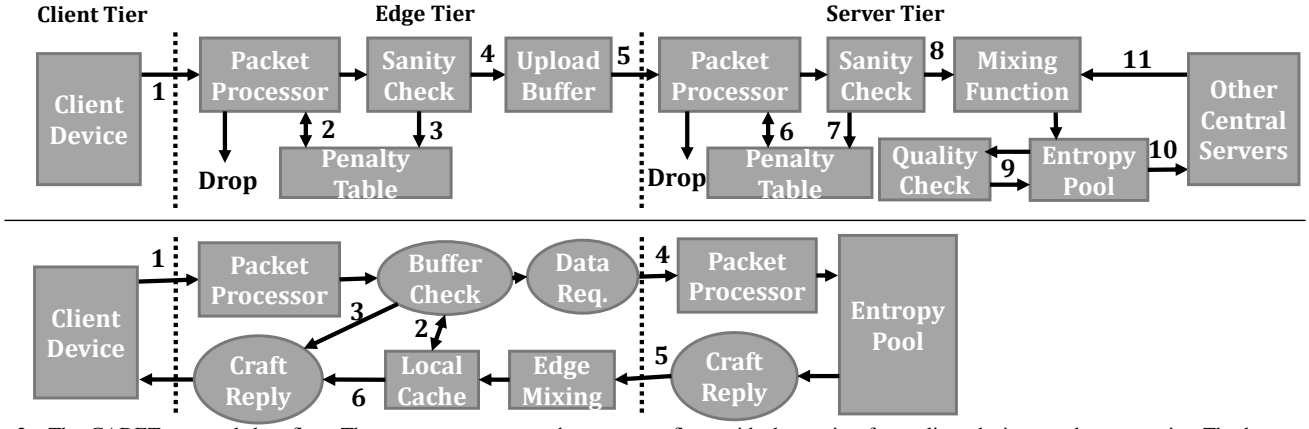


Fig. 2. The CADET protocol data flow. The top row represents the upstream flow, with data going from client devices to the server tier. The bottom row represents the downstream flow, with data coming from the server tier down to client devices.

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

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

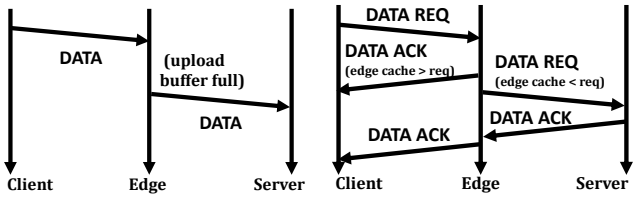
III. CADET Data Transport

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 2 illustrates the data flow architecture.

A. CADET Transport Design

As discussed in Section II, devices participating in the protocol are organized in a distributed, tree-like hierarchy. 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 2 illustrates the flow of entropy data from clients to the server tier, while Figure 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 module. 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.



(a) Data Upload (b) Data Request

Fig. 3. Basic packet exchange timelines for data uploads and data requests in the CADET framework.

0	Version Number					Reserved		
8	REG	DAT	REQ	ACK	C-E	E-S	ENC	URG
16	Variable Arguments							
24	Variable Arguments							
32	Data Payload							

Fig. 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.

Data Request: The bottom half of Figure 2 illustrates when a client makes a request for additional entropy, while Figure 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 finally responds to the client's request (6).

B. CADET Packet Structure

Figure 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).

C. 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 1.

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

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 [20]. Empirically, we choose a decay value of 0.96 and a threshold of 3 standard deviations above the mean usage score.

IV. 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 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.

A. 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 5. The number listed beneath the figure represent a user's penalty score, and increases left to right.

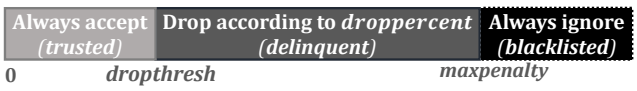


Fig. 5. The CADET protocol drop strategy for sanity checks. User penalty increases left to right up to some threshold.

A device's penalty ranges from $[0, \infty)$. Points are removed or added depending on the quality of the data according to

TABLE I
SANITY CHECK PENALTY SCHEMES

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

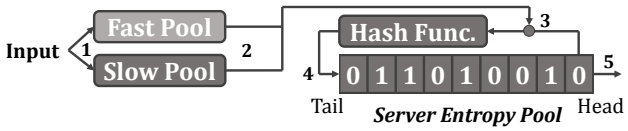


Fig. 6. The CADET protocol mixing function, which serves to combine incoming data at the server level before storing it for future use.

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 2. Alternative equations for `drop_percent`, such as the sigmoid function, can also be used in order to avoid a 100% drop rate. The penalty scheme used in the prototype implementation of CADET is in Table I, along with other potential alternatives, as different edge nodes may have different requirements.

$$drop\_percent = \frac{userpenalty - dropthresh}{maxpenalty - dropthresh} \quad (2)$$

To implement sanity checks in the system, we've taken 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 [21]. Each of these tests are computationally light, requiring only one or two passes through the data, keeping the amount of processing per bit linear.

B. Mixing Function

In the CADET architecture, the mixing function directly impacts the output quality by how well it folds together 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 [13]. 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 our mixing function in Figure 6. Data 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.

C. 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 [21]. 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.

V. CADET DATA SECURITY

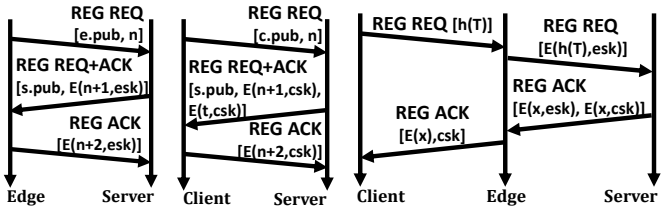
Data security is the process of ensuring that a client's data or service quality is not affected by a malicious entity. This means both the contents of the data as well as the delivery of the data itself. We focus on three main threat vectors in the scope of this work - *service degradation*, *quality 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 consumption 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.

A. 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 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



(a) Edge Reg. (b) Client Init. (c) Client Reregistration

Fig. 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).

`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.

B. 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 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).

C. 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 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 forwards the client half of the encrypted payload, after which both the edge and client devices decrypt and obtain the shared key `cek`.

VI. 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.

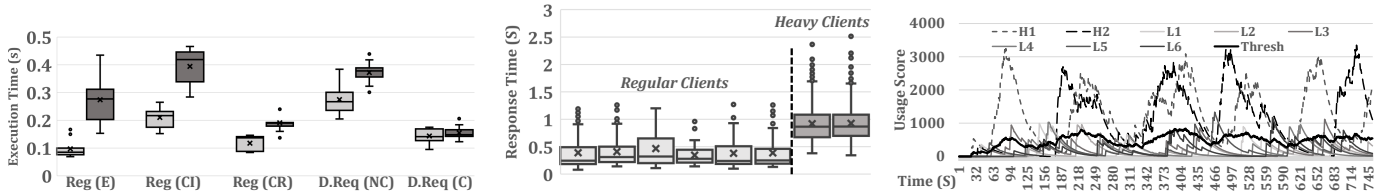
A. Testbed Setup

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 [22]. The topology for this network is shown in Figure 9. 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.

We have underclocked each Pi according to the labels in Figure 9, 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.

B. Data Transport

1) Protocol Timing: Here, we measure the window from the moment the first packet leaves the source device to after all processing for the final packet has been resolved. The results of this experiment are summarized in Figure 8a. In



(a) Protocol Operations Timing (b) Edge Response Time During Heavy Use (c) Usage Score over Time

Fig. 8. a) Execution time for different actions including travel time. The left and right boxes show the testbed (no internet) and real world timings respectively. Registration (Reg) of Edge (E), Client Init (CI) and Client Rereg. (CR). Data Request (D.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. c) Usage score over time with a network of two heavy users (dashed lines) and six light (solid lines) users.

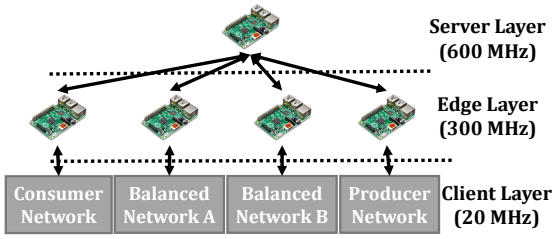


Fig. 9. The Raspberry Pi testbed topology. Each "network" box in the client tier represents 11 Raspberry Pi devices under a particular edge node, connected via a single switch. The clock speed for each tier is listed beside the tier name.

general, response times are very low, 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.

2) Edge Node Effect: 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 10a and 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 (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 (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.

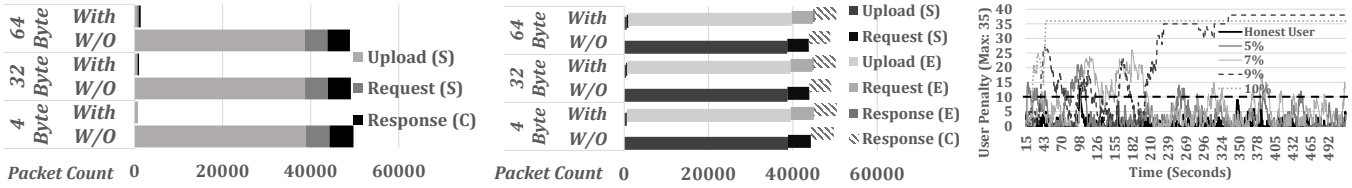
3) Usage Score: We orchestrated one network of 8 Raspberry Pi's to investigate how well the usage score can identify heavy users. We plotted the usage score over time of all devices, two of which were intentionally tuned to be heavier users. Figure 8c shows the results of one data trace. While this is 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.

C. Data Quality

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 10c shows these results. Under the default CADET penalty scheme, a clients 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 V), it is possible to push these numbers higher or lower on a per-edge node basis.

Table II 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 $\leq$ 3 checks passed) should be dropped. For clients who upload 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. 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.

2) Quality Checks: To evaluate the quality of the mixing function, we use the NIST statistical test suite on the data



(a) Server Load Reduction by Edge (b) Network Cost of Edge (c) User Penalty Plot

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

TABLE II
SANITY CHECK ACCURACY VS. CLIENT BEHAVIOR

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 III
P-VALUES FOR QUALITY ASSURANCE TESTS

	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

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 for each test, but in general a higher value indicates a stronger suggestion of randomness, and p must be above 0.01 [21]. For comparison, we show our values against those produced by running the suite on the Linux PRNG [4]. Table III 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 [6].

D. Data Security

Due to space, the authors cannot give a 100% comprehensive security evaluation of the CADET protocol. However, we use this section to highlight some of the more obvious threat vectors and how we mitigate them. We assume that all participating devices are not compromised - an attacker does not have control of, or the ability to read data within a device. We consider three different threat models: eavesdropping, service degradation, and randomness degradation.

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 [23], [24]. 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.

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 device 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 8b, 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.

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.

VII. Related Work

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 [16][3]. 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 [17]. 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 [25]. 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 [26]. 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 [27]. Only in recent years has there been a growth of true EaaS services, such as netRandom [28]. 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.

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 [7], predictable TLS keys in virtual environments [12], predictable OpenSSL keys on Android [29], predictable initial RNG outputs [11], 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.

Recent efforts have tried to standardize software used across low-profile IoT devices to improve interoperability. Google has proposed their Android Things™ platform as a standard executing environment for IoT devices [30]. Similarly, the Google Weave™ communication protocol allows for these devices to more easily communicate through a unified language [31]. Other instances of unified protocols and platforms exist, such as Mozilla's *Things Gateway* [32], Open Habitat [33], or Home Assistant [34]. 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 CADET.

VIII. Discussion and Future Work

We have constructed a working prototype of our proposed Collaborative and Distributed Entropy Transfer protocol. However, there are many questions and features that were unable to be explored in the scope of this paper. Here, we briefly summarize potential areas for exploration further research.

First is deeper analysis of supply and demand. As the number of devices 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. At the current stage, we make the assumption that the edge node can be 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 coffee 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 to it by participants. However, clients who contribute above a certain threshold may wish to be compensated. Similarly, building up a network of central servers requires hardware and bandwidth. Potential incentive models include public utilities like electricity (e.g. users with solar panels), or sharing economy systems (e.g. ride sharing).

IX. 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.

Acknowledgment

The authors thank Ed Novak for his coding contributions to this project, as well as members of the LENS lab for their feedback. 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.

References

[1] R. H. Weber, "Internet of things–new security and privacy challenges," *Computer law & security review*, vol. 26, no. 1, pp. 23–30, 2010.

[2] K. Zhao and L. Ge, "A survey on the internet of things security," in *Computational Intelligence and Security (CIS), 2013 9th International Conference on*. IEEE, 2013, pp. 663–667.

[3] M. Haahr, "Random.org - true random number service," http://random.org, 1998.

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

[5] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the linux random number generator," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 15–pp.

[6] A. Vassilev and R. Staples, "Entropy as a service: Unlocking cryptography's full potential," *Computer*, vol. 49, no. 9, pp. 98–102, 2016.

[7] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. 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)*. Bellevue, WA: USENIX, 2012, pp. 205–220. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger

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

[9] J. Cheetham, "Onerng - open hardware random number generator," http://onerng.info, 2014.

[10] ——, "Truerng - hardware random number generator," http://ubld.it/products/truerng-hardware-random-number-generator/, 2013.

[11] A. Everspaugh, Y. Zhai, R. Jellinek, T. Ristenpart, and M. Swift, "Not-so-random numbers in virtualized linux and the whirlwind rng," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 559–574.

[12] T. Ristenpart and S. Yilek, "When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography." in *NDSS*, 2010.

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

[14] P. Gopalan, R. Meka, O. Reingold, L. Trevisan, and S. Vadhan, "Better pseudorandom generators from milder pseudorandom restrictions," in *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*. IEEE, 2012, pp. 120–129.

[15] K. Wallace, K. Moran, E. Novak, G. Zhou, and K. Sun, "Toward sensor-based random number generation for mobile and iot devices," *IEEE Internet of Things Journal*, 2016.

[16] J. Walker, "Hotbits - genuine random numbers," https://www.fourmilab.ch/hotbits/, 1996.

[17] M. Wood and G. Graunke, "Enhancing entropy in pseudo-random number generators using remote sources," Mar. 30 2001, uS Patent App. 09/822,548.

[18] J. R. Enzminger, "Method, apparatus, and program product for distributing random number generation on a gaming network," Mar. 4 2014, uS Patent 8,662,994.

[19] J. Niset and L.-P. Lamoureux, "Random number distribution," Dec. 5 2014, uS Patent App. 14/562,308.

[20] J. Kurose, *Computer Networking*, 6th ed. Pearson, 2013.

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

[22] R. P. Foundation, "Raspberry pi resources," https://raspberrypi.org, 2014.

[23] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *International Workshop on Public Key Cryptography*. Springer, 2006, pp. 207–228.

[24] M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe, "High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers," *Designs, Codes and Cryptography*, vol. 77, no. 2-3, pp. 493–514, 2015.

[25] A. Toponce, "True random number service - entropy as a service," http://hundun.ae7.st/, 2012.

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

[27] D. Kirkland, "Ubuntu pollen," https://github.com/dustinkirkland/pollen, 2014.

[28] W. E. S. Inc., "Quantum entropy-as-a-service," https://www.getnetrandom.com/, 2017.

[29] S. H. Kim, D. Han, and D. H. Lee, "Predictability of android openssl's pseudo random number generator," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 659–668.

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

[31] ——, "Google weave," https://developers.google.com/weave/, 2017.

[32] Mozilla, "Things gateway," http://iot.mozilla.org/gateway/, 2017.

[33] O. Foundation, "Openhab," https://www.openhab.org, 2016.

[34] H. Assistant, "Home assistant," https://homeassistant.io, 2017.

[35] D. Kaplan, S. Kedmi, R. Hay, and A. 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: USENIX Association, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/kaplan

[36] M. Smith, "Peeping into 73,000 unsecured security cameras thanks to default passwords," 2014. [Online]. Available: http://www.networkworld.com/article/2844283/microsoft-subnet/peeping-into-73-000-unsecured-security-cameras-thanks-to-default-passwords.html

[37] G. Marsaglia, "Diehard battery of tests of randomness," 1995.

[38] G. J. Chaitin, *Information, randomness & incompleteness: papers on algorithmic information theory*. World Scientific, 1990, vol. 8.