entropy

MDPI

Article

Recoverable Random Numbers in an Internet of Things Operating System

Taeill Yoo [1], Ju-Sung Kang [1,2] and Yongjin Yeom [1,2,*]

[1] Department of Financial Information Security, Kookmin University, Seoul 02707, Korea; taeillyoo@kookmin.ac.kr (T.Y.); jskang@kookmin.ac.kr (J.-S.K.)

[2] Department of Mathematics, Kookmin University, Seoul 02707, Korea

* Correspondence: salt@kookmin.ac.kr; Tel.: +82-2-910-5749

Academic Editor: Kevin H. Knuth

Received: 29 December 2016; Accepted: 9 March 2017; Published: 13 March 2017

Abstract: Over the past decade, several security issues with Linux Random Number Generator (LRNG) on PCs and Androids have emerged. The main problem involves the process of entropy harvesting, particularly at boot time. An entropy source in the input pool of LRNG is not transferred into the non-blocking output pool if the entropy counter of the input pool is less than 192 bits out of 4098 bits. Because the entropy estimation of LRNG is highly conservative, the process may require more than one minute for starting the transfer. Furthermore, the design principle of the estimation algorithm is not only heuristic but also unclear. Recently, Google released an Internet of Things (IoT) operating system called Brillo based on the Linux kernel. We analyze the behavior of the random number generator in Brillo, which inherits that of LRNG. In the results, we identify two features that enable recovery of random numbers. With these features, we demonstrate that random numbers of 700 bytes at boot time can be recovered with the success probability of 90% by using time complexity for 5.20×2^{40} trials. Therefore, the entropy of random numbers of 700 bytes is merely about 43 bits. Since the initial random numbers are supposed to be used for sensitive security parameters, such as stack canary and key derivation, our observation can be applied to practical attacks against cryptosystem.

Keywords: Linux Random Number Generator; random number recovery; entropy source; Brillo; Internet of Things (IoT) operating system

1. Introduction

Important secret variables, such as encryption keys, salts, generation of primes, and stack canaries, are generated by a random number generator (RNG). The problem with RNGs is that they can leak this important information to attackers. In recent decades, this problem has been revealed in various platforms ranging from early versions of Netscape's secure sockets layer (SSL) [1] to smartphone environments (e.g., Androids). If weak random numbers are generated, the private key can be illegally recovered in the public key cryptosystem [2–4]. Furthermore, predictable random numbers can be generated if the RNG has insufficient noise sources, such as boot time.

For example, several articles [5–8] pointed out that the important parameters (e.g., PreMasterSecret) can be exposed in the embedded system, Androids, and OpenSSL from predictable random numbers generated at boot time because collecting noise sources is limited. In addition, the Bitcoin wallet was attacked in the elliptic curve digital signature algorithm (ECDSA) process because the Java-based RNG (SecureRandom class) is vulnerable [9]. Recoverable random numbers were also leveraged in a backdoor in the international standard of the dual elliptic curve deterministic random bit generator (Dual EC DRBG) [10]. Checkoway et al. [11] showed that utilizing the backdoor is of practical use. A systematic analysis on the security of linux random number generator (LRNG) was initiated by

Guttermann et al. [12]. In their work, a concrete structure and the entropy collecting process from noise sources in LRNG were investigated. It was additionally shown that 2^{96} complexity is required to restore the state of the entropy pool from generated random numbers. Recently, Heinger et al. [3] studied the internet-wide vulnerability of RNG. In transport layer security (TLS) as well as secure shell (SSH), numerous certificates were collected and several identical keys were found by extracting common primes using the greatest common divisor (GCD) algorithm. Moreover, Kim et al. [7] showed that PreMasterSecret can be recovered by 2^{52} complexity in the handshake process of OpenSSL in the Android. Their attacks can be possible because predictable random numbers are generated from insufficient entropy at boot time. Kaplan et al. [8] attacked the Android KeyStore using a stack buffer-overflow vulnerability (CVE-2014-3100) [13] by leveraging the above-mentioned RNG problem. For CVE-2014-3100 to succeed, it must bypass the stack canary that serves as the defense for a stack buffer-overflow attack. However, the stack canary can be predictable, provided that it is generated by RNG along with the mentioned problem.

The underlying cause of these results can be divided into two features. First, when random numbers are requested from the non-blocking pool via /dev/urandom or get_random_bytes(), there is no entropy transfer from the input pool to the non-blocking pool if the entropy counter of the input pool does not reach the threshold (192 bits). This means that the process of generating random numbers in the non-blocking pool is deterministic. The second is that estimating entropy of LRNG is conservative. Consequently, noise sources are inefficiently used and it has to spend a lot of time for harvesting entropy from noise sources until the entropy counter exceeds the transfer threshold (192 bits). Moreover, without entropy transfer [6], it makes generating random numbers from the non-blocking pool deterministic. In particular, because of limited noise sources in the embedded environments, this feature is closely related to LRNG security. In addition, as observed in [7,8], the predictability of random numbers generated at the initial boot time can be feasible because it is not easy to collect sufficient noise sources from booting.

In November 2015, Google released the source code for an Internet of Things (IoT)-specific operating system called Brillo [14]. It is evident in the source code that Brillo is based on the Linux kernel (version 3.10); moreover, the structure of RNG is identical to the previous version. This implies that Brillo still has problems identical to those of LRNG. To verify this point, we analyze the behavior of LRNG from the boot start. According to the results, we conclude that the entropy counter in the input pool at boot time does not exceed the threshold (192 bits), and the sequence is consistent for collecting noise sources (input) and generating random numbers (output). Experimentally, for random numbers of 700 bytes during boot time, we determine that it can be recovered with the probability of 90% within the cost of 5.20×2^{40}. Some of the 700 bytes are used as a stack canary to prevent a stack buffer-overflow attack, which is a potential vulnerability.

2. Structure of LRNG and Brillo

2.1. LRNG Structure

LRNG [15] has three entropy pools, which are referred to as the input pool and output pools (blocking pool and non-blocking pool). The size of the input pool is 4096 bits (128 words = 4 bytes $\times$ 128) and the size of the output pool is 1024 bits (32 words = 4 bytes $\times$ 32). In Linux, five noise sources (disk Input/Output (I/O), human input, interrupt, device information, and hardware RNG) are available in the input pool. The entropy of human input, such as a mouse click or a keyboard stroke, and disk I/O noise is estimated by the difference between the times in which an event occurs. The entropy of the interrupt is always estimated as one bit. The device information is not reflected in the entropy counter. The hardware RNG operates as a noise source, if it is available. Its entropy is reflected as offline-measured values. However, hardware RNG is unavailable in Brillo because Brillo is based on kernel version 3.10, whereas hardware RNG is supported from version 3.17 onwards. Because the noise source generated in the boot process mainly consists of device information and interrupt, the entropy

counter of the input pool is not changed or increased by one bit in most cases. Therefore, the entropy counter of the input pool slowly increases during the boot process. After estimating entropy, collected data from the noise sources are accumulated in the input pool through the mixing function.

There are two ways to generate random numbers from these entropy pools (blocking pool and non-blocking pool). /dev/random is used to produce random numbers from the blocking pool, and /dev/urandom or get_random_bytes() is used to generate random numbers from the non-blocking pool. The difference between the two methods lies in the usage of the entropy counter. In the case of the blocking pool, the entropy is transferred up to the requested size until the entropy counter is sufficiently large to generate secure random numbers. On the other hand, random numbers are immediately generated from the non-blocking pool regardless of its entropy counter. Figure 1 depicts the procedure and building blocks of LRNG.

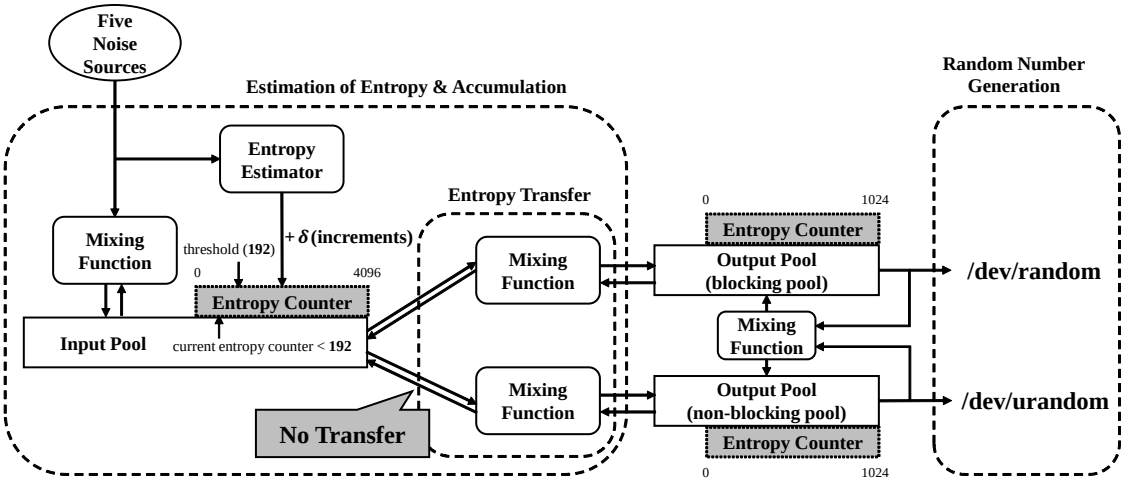


Figure 1. Relations between three entropy pools in LRNG.

Most of the random numbers are produced through the get_random_bytes() function. This function internally uses Secure Hash Algorithm 1 (SHA-1) and LFSR-based mixing function. First, output of the 160 bits is generated from the non-blocking pool through SHA-1. Then, it is again injected through the mixing function into the non-blocking pool. Next, 512 bits of the non-blocking pool and previously generated output of 160 bits are entered into SHA-1. Then, output of 160 bits is generated. This output is split into the most significant 80 bits and the least significant 80 bits, which are then eXclusive-OR (XOR)-ed. Finally, get_random_bytes() generates 80 bits of random numbers [15]. This extraction process is referred to as an *extract unit* when the cost of recovery is calculated (Figure 2).

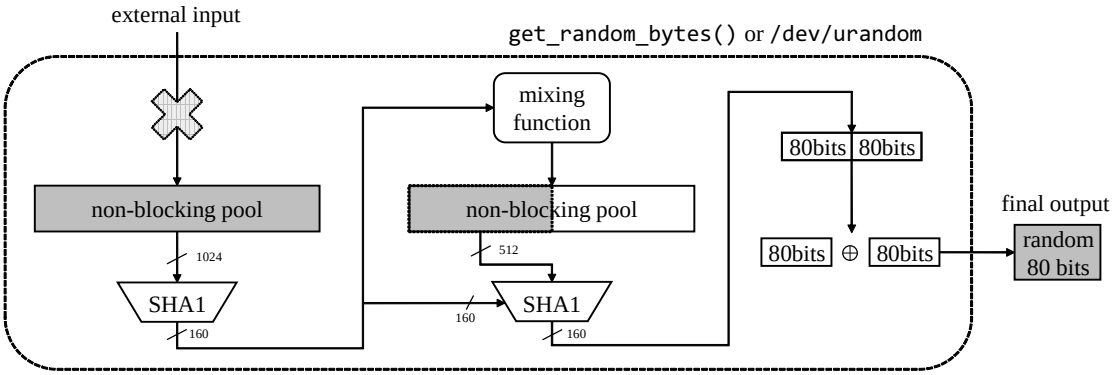


Figure 2. *Extract unit*: Extracting random numbers using get_random_bytes() or /dev/urandom without the entropy transfer.

This structure causes several issues such as the above-mentioned problems. In particular, generating random numbers by get_random_bytes() is deterministic in restricted environments such as embedded systems, since collecting sufficient noise sources is extremely hard. In this case, if the state of the non-blocking pool is compromised, the attacker can obtain all random numbers at a particular point in time. As Brillo uses LRNG without changes (kernel 3.10), Brillo has identical problems to LRNG.

2.2. Analysis Points of Brillo

This section briefly describes the structure of Brillo [16] according to the boot sequence. We check three points of analysis to find closely related parts of RNG. The boot loader loads the Linux kernel image when the power is turned on. Next, the loaded kernel proceeds to the initialization process (start_kernel()) and executes the init process. The init process is finished after running daemons and libraries. In the case of the Android, the runtime library, framework, and application layer are loaded [17]. However, in the case of Brillo, these layers are removed to fit the IoT platform. Therefore, there is no virtual machine (e.g., Dalvik Virtual Machine (VM), Android Runtime) that interprets the Java language. The application layer is only implemented in C/C++ native code. Accordingly, three points are associated with LRNG in the structure of Brillo:

- Kernel initialization: LRNG is initialized. All the entropy pools are initialized and filled with zeros.
- Init process: All daemons and libraries are initialized.
- Cryptographic library: Each library calls LRNG in its own RNG (e.g., OpenSSL, BoringSSL, and Weave).

To analyze how LRNG operates from boot time in Brillo, we first start the analysis by focusing on the kernel initialization phase. Figure 3 indicates the structure and points of analysis in Brillo.

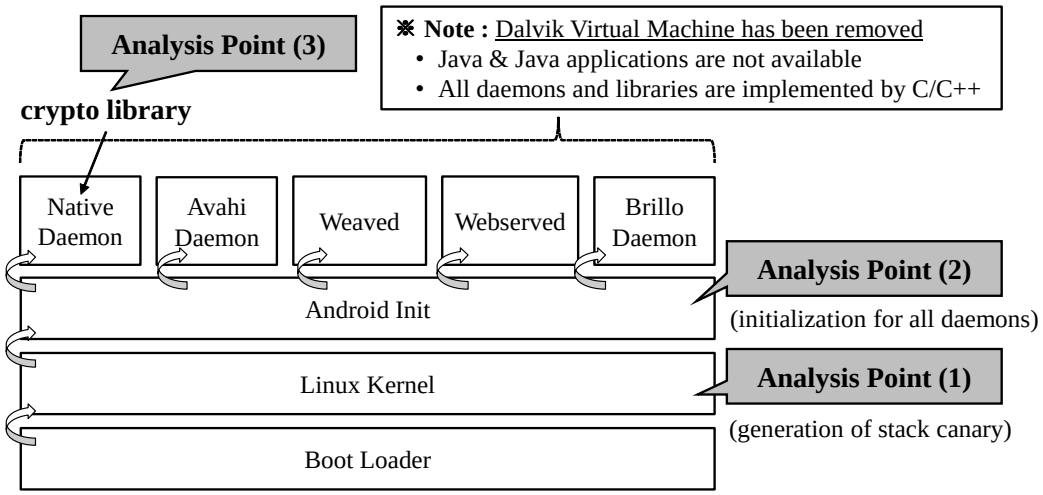


Figure 3. Structure of Brillo and three analysis points.

3. Analyzing Features of LRNG during Boot Process

3.1. Strategy and Environments for Analysis

The goal of this analysis is to identify the behavior of LRNG during boot time of Brillo. Through this analysis, we collect the necessary data, such as values of noise source, entropy counters, and sequences of input and output. The analysis process is divided into seven steps (Figure 4). The analysis is conducted at the Edison board (Arduino Kit) [18]. Edison board was the only released board when we started our analysis. In 2017, four development boards are available (http://elinux.org/Android_Brillo_Internals). The kernel is compiled with GCC 4.9 version in Ubuntu 14.04 Long Term Support (LTS). The files

created by compiling the kernel are ported using Intel Flash Phone Tool Lite (5.3.2.0) (Table 1). Other necessary settings follow the instructions of [19].

Table 1. Files created for porting after compiling.

Type	File Name
.img	Boot.img, cache.img, ramdisk.img, recovery.img, system.img, u-boot-edison.img, userdata.img
.bin	edison-dnx-fwr.bin, edison-dnx-osr.bin, edison-ifwi-dbg-00.bin, edison-ifwi-dbg-01.bin, edison-ifwi-dbg-02.bin, edison-ifwi-dbg-03.bin, edison-ifwi-dbg-04.bin, edison-ifwi-dbg-05.bin, edison-ifwi-dbg-06.bin, gtp.bin, u-boot-edison.bin
scrypt	FlashEdison.json

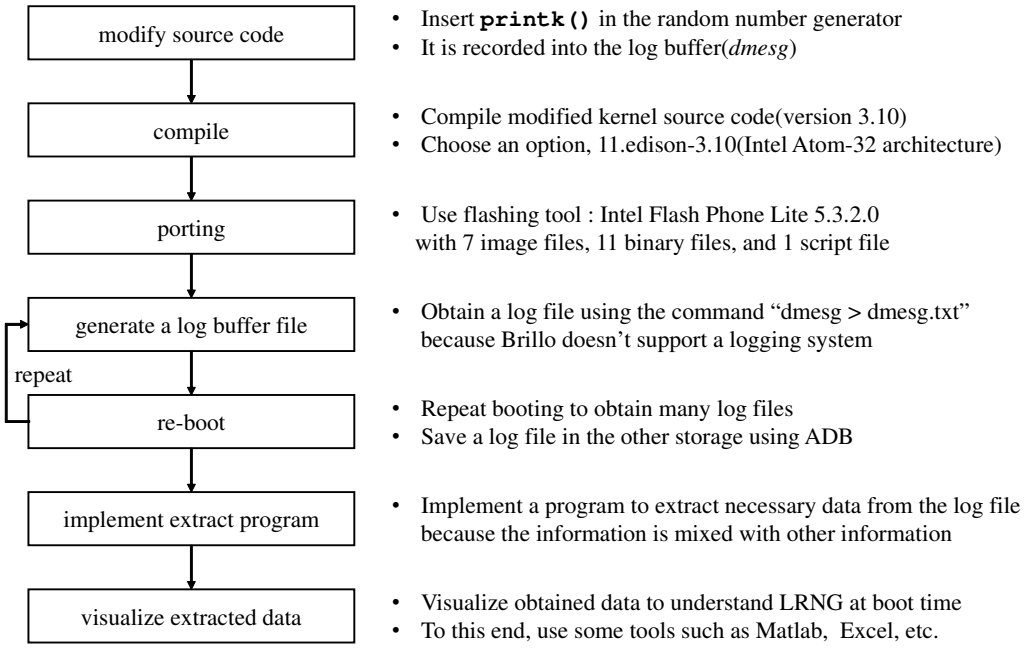


Figure 4. Flow chart of analysis process.

The path of source code of LRNG is located in /dev/char/random.c. First, we insert several codes (e.g., printk()function) in the random.c to check behavior of LRNG in the boot process. From these codes, the necessary information is recorded in the boot log.

In the case of Linux and Android, a logging system exists to record all boot processes. All boot records are stored in a file (syslog). However, Brillo does not currently support the logging system. Instead, the boot log is saved in the *dmesg* ring buffer of 64 KB. Therefore, it must move them to the non-volatile storage before the power is turned off. We connect to Brillo using Android Debug Bridge (ADB). By repeating the boot, we obtain many boot logs and extract necessary information.

3.2. Two Features of Generating Random Numbers in Brillo

By analyzing the behavior of LRNG during the boot process in Brillo, we found two features for generating random numbers. Using these two features, it is possible to recover random numbers generated at boot time within a practical time frame. The first feature is that the entropy counter of the input pool is less than the transfer threshold (192 bits). When this condition is satisfied, the process of generating random numbers is almost deterministic because there is no entropy transfer. The second

feature is that the pattern of input and output is consistent for each boot process. Owing to this feature, the cost of recovery is considerably reduced.

3.2.1. First Feature: Entropy Counter of the Input Pool at Boot Time is Less than 192 Bits (Insufficient Entropy)

We insert printk() to record the entropy counter in the kernel source code before collecting noise sources and after producing random numbers. Then, the modified kernel is booted 10,000 times, thereby producing 10,000 graphs of the entropy counter. These graphs have a similar pattern. It involves an average of 28 s to complete boot up. Figure 5 indicates a typical graph for the entropy counter. The entropy counter is less than 192 bits during boot time. Therefore, generating random numbers by /dev/urandom or get_random_bytes() is "almost" deterministic because there is no entropy transfer from the input pool to the non-blocking pool.

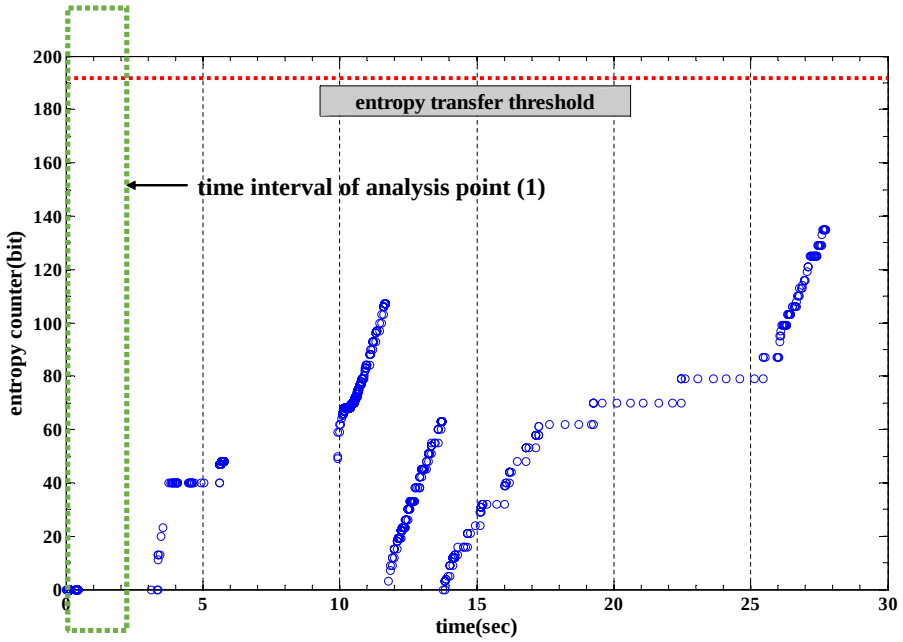


Figure 5. Entropy counter of the input pool during boot time in Brillo.

The expression "almost deterministic" means that there is an exceptional noise source directly injected into the non-blocking pool even though all noise sources are accumulated in the input pool. Owing to this noise source, the state of the non-blocking pool is not fully deterministic. Slight randomness occurs from this exceptional noise source. This noise source serves to inject the device-specific information (add_device_randomness()), which is mainly inputted at boot time. This is assumed to prevent the deterministic state for the non-blocking pool during boot time.

Figure 6 indicates the code of add_device_randomness(). For direct input in the non-blocking pool, *buf*, time(*cycles*, *jiffies*) are used. *buf* is a fixed value, such as device name and serial number. *jiffies* is also fixed values in the analysis point (1). Therefore, *cycles* is the only noise source for analysis. If the entropy counter is less than 192 bits, the randomness of the non-blocking depends only on *cycles*. Even though add_device_randomness() provides random data directly injected into the non-blocking pool, it is not reflected by the entropy counter.

```
1. add_device_randomness(char* buf, int num)
2. {
3.     time = cycles ^ jiffies;
4.     mix_bytes_pool(blocking, buf);
5.     mix_bytes_pool(blocking, time);
6.     // buf is directly injected into the nonblocking pool
7.     mix_bytes_pool(nonblocking, buf);
8.     // time is also directly injected into the nonblocking pool
9.     mix_bytes_pool(nonblocking, time);
10. }
```

Figure 6. Code of add_device_randomness().

Note that similar phenomena are observed in the Android environments [6] as well. In [6], the authors analyzed the entropy counter of the boot time in some Android smartphones such as Nexus 4, Nexus 7, and Galaxy Nexus. They found that the entropy counters are quickly increased before Android smartphones finish the boot sequence. The difference of hardware devices causes the different result. Android mainly uses disk I/O as the entropy source. Once LRNG collects entropy from this source, its entropy estimator calculates the entropy and increases the entropy counter. As a result, the entropy counter reaches the threshold quickly and the entropy is transferred into the non-blocking pool. However, Brillo only uses entropy from device information and time, which does not increase the entropy counter at all, because Brillo does not have a disk, keyboard, or a mouse.

3.2.2. Second Feature: Order of Inputting Noise Source and Outputting Random Numbers are Consistent (Identical Pattern)

For the non-blocking pool, the identical pattern means that the sequences of noise source input and random number output are consistent. When the timing of input and output are recorded in a timeline, the time axis can be divided into several sections based on the time stamps of *input*. Figure 7 represents the timeline for the sequence of input and output. The following notations indicate indices for *input*, *output*, and *interval*.

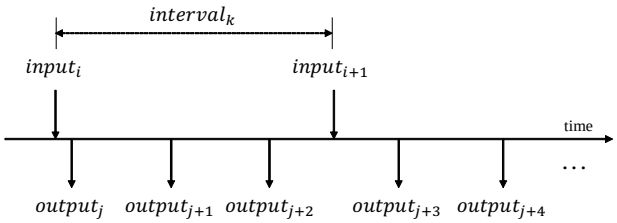


Figure 7. Timeline for input and output timing.

- $input_i$: i-th input entropy source, $i = 1, 2, \cdots$
- $output_j$: j-th output random number, $j = 1, 2, \cdots$
- $interval_k$: a section between $inputs$, $k = 1, 2, \cdots$

When the boot starts, the three entropy pools of LRNG are initialized to all zeros in the kernel initialization phase. The first random numbers (8 bytes) are produced from this state. Next, the first noise source (add_device_randomness()) is collected as input. Because this noise source is device information, it is directly injected into the non-blocking pool. Then, subsequent random numbers of 24 bytes are produced three times (8 bytes $\times$ 3). Then, a noise source (device information) is injected again and random numbers of 668 bytes are consecutively produced 86 times (Table 2).

In the boot process, this sequence is consistent until the 101st random numbers are produced (identical pattern). It is maintained until approximately the 4.6 s point. After this time, it is difficult

to predict the in–out sequence because the operating system is switched to multi-core environments (i.e., race condition). As the Edison board has a dual-core CPU, another CPU starts to access LRNG from this point.

Table 2. Index, size, and usage for the sequence of the identical pattern.

Interval	Output	Size (Bytes)	Usage	Identical Pattern
1	1	8	stack canary	
2	2	8	stack canary	
	3	8	stack canary	
	4	8	stack canary	
3	5–84	640 (= 8 × 80)	unknown	
	85–88	16 (= 4 × 4)	unknown	
	89	4	unknown	
	90	8	unknown	
...				yes
15	91–94	16 (= 4 × 4)	unknown	
...				
26	95	6	unknown	
	96	4	unknown	
	97	136	unknown	
	98–101	16 (= 4 × 4)	unknown	
27	102–104	12 (= 4 × 3)	unknown	
	105–106	24 (= 12 × 2)	unknown	
28	107	8	unknown	
	108	16	unknown	no
	109	4	unknown	
	110	64	unknown	
	111	16	unknown	

Table 2 summarizes the identical pattern in the timeline of Figure 8. Meanwhile, $output_1$, $output_2$, $output_3$, and $output_4$ become stack canaries, while the remaining values continue to be analyzed.

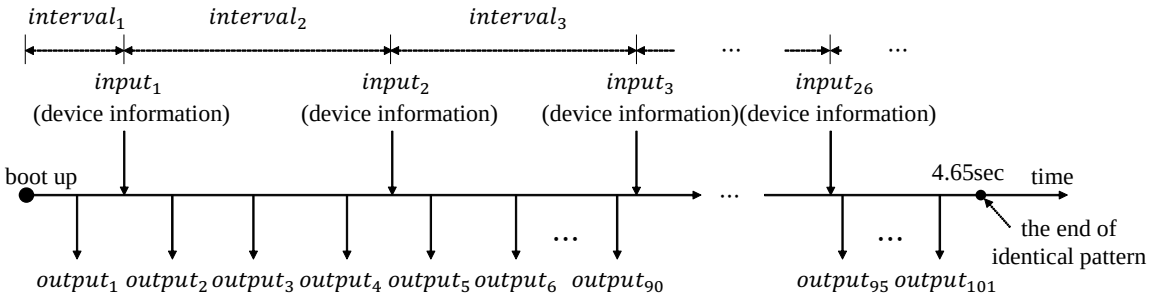


Figure 8. Timeline for identical pattern at boot time in Brillo.

The first feature (insufficient entropy) is unique to LRNG. This feature is also satisfied in Brillo and makes LRNG generate random numbers without any entropy transfer. The second feature (identical pattern) is a unique one that is observed during boot time of Brillo. By combining the two features, recovering random numbers between $interval_1$ and $interval_3$ depends on *cycles*. Therefore, *cycles* is the most important factor to recover random numbers in these intervals.

3.3. Success Probability of Recovery

Cycles is the only noise source, which is directly injected into the non-blocking pool, to generate random numbers (e.g., /dev/urandom or get_random_bytes()) at boot time. Even though the boot process is routine, the values of *cycles* is somewhat random. The randomness of *cycles* comes from some frequency differences between hardware devices such as processor, cache memory, disk, etc. In the aspect of an attacker, observing these states is very difficult. However, the values of *cycles* can be observable; thus, we regard *cycles* as a random variable. By modeling the distribution for *cycles*, we obtain the attacker's success probability of recovery and the cost of attack. In this subsection, we focus on random numbers of 700 bytes in $interval_1$, $interval_2$, and $interval_3$ and describe probabilistic models for $cycles_1$ and $cycles_2$.

For $cycles_1$ and $cycles_2$, we collected sample values of 10,000 by iterative bootings. From their histograms, the distributions are estimated as exponential distributions.

3.3.1. Probabilistic Models of $Cycles_1$ and $Cycles_2$

A random variable whose probability density function f is given by, for some $\lambda > 0$,

$$f(x) = \lambda e^{-\lambda x}, x \geq 0$$

is called an exponential random variable with parameter λ. The cumulative distribution of X, the exponential random variable with parameter λ, is given by

$$Pr[X \leq \alpha] = 1 - e^{-\lambda\alpha}, \alpha > 0, \text{ and } E[X] = \frac{1}{\lambda}.$$

Because an arrival time of the first event follows an exponential distribution, *cycles* also can be regarded as an arrival time when a noise source is injected into the entropy pool (non-blocking pool). The exponential distribution is used to describe the time between events in a Poisson process in which events occur continuously and independently at a constant average rate [20]. Thus, we can reasonably assume that *cycles* is an exponential random variable for some appropriate value of λ. In order to lead the value of λ, we estimate the expectation of the random variable X by the average of several sample values, and we apply the fitting process of MATLAB [21] (version R2016b) for converting a histogram to the probability density function of X:

$$supp(X_1) = \overline{\{x_1 : Pr[X_1 = x_1] > 0\}} \approx [2803, 2867],$$
$$supp(X_2) = \overline{\{x_2 : Pr[X_2 = x_2] > 0\}} \approx [3513, 3577].$$

Let X_1 and X_2 denote two random variables of $cycles_1$ and $cycles_2$, respectively. From sample values collected by iterative bootings, we consider the dominant parts of supports of X_1 and X_2. For a random variable X, the support is defined by the closure of the set containing all possible values x of X such that $f_X(x) > 0$, $supp(X) = \overline{\{x : f_X(x) > 0\}}$, where f_X is the probability density function of X. Then, the supports of X_1 and X_2 are estimated as follows:

$$supp(X_1) \approx [2803, 2867], \quad supp(X_2) \approx [3513, 3577].$$

From Figure 9, we can obtain the fact that $supp(X_1)$ and $supp(X_2)$ are clearly non-overlapped. Thus, we suppose that two random variables X_1 and X_2 are independent. In fact, for our 10,000 sample values of $cycles_1$ and $cycles_2$, we have obtained that $Pr[X_1 \leq \alpha]Pr[X_2 \leq \beta] \approx Pr[X_1 \leq \alpha, X_2 \leq \beta]$, for several $\alpha, \beta > 0$. By shifting the starting points of X_1 and X_2 to zero, we define the corresponding random variables Y_1 and Y_2 by $Y_1 = X_1 - 2803$ and $Y_2 = X_2 - 3513$, respectively. Then, Y_1 and Y_2

are modelized by exponential distributions with parameter $\lambda_1 > 0$ and $\lambda_2 > 0$, respectively. We can estimate the values of λ_1 and λ_2 from the 10,000 sample values by

$$\frac{1}{\lambda_1} = E[Y_1] = E[X_1] - 2803 \approx 0.595898,$$
$$\frac{1}{\lambda_2} = E[Y_2] = E[X_2] - 3513 \approx 0.709255.$$

That is, $\lambda_1 \approx 1.68$ and $\lambda_2 \approx 1.41$. By using distribution fitting tool of MATLAB, we confirm several properties of the distributions. Fitting results show that the sample values converge on the exponential distributions (Figure 10).

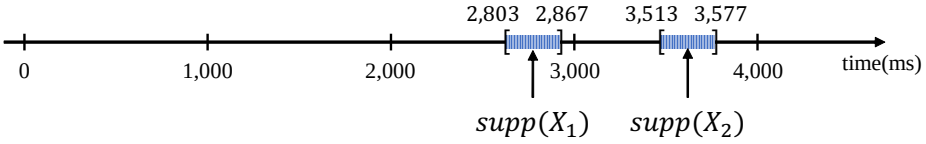
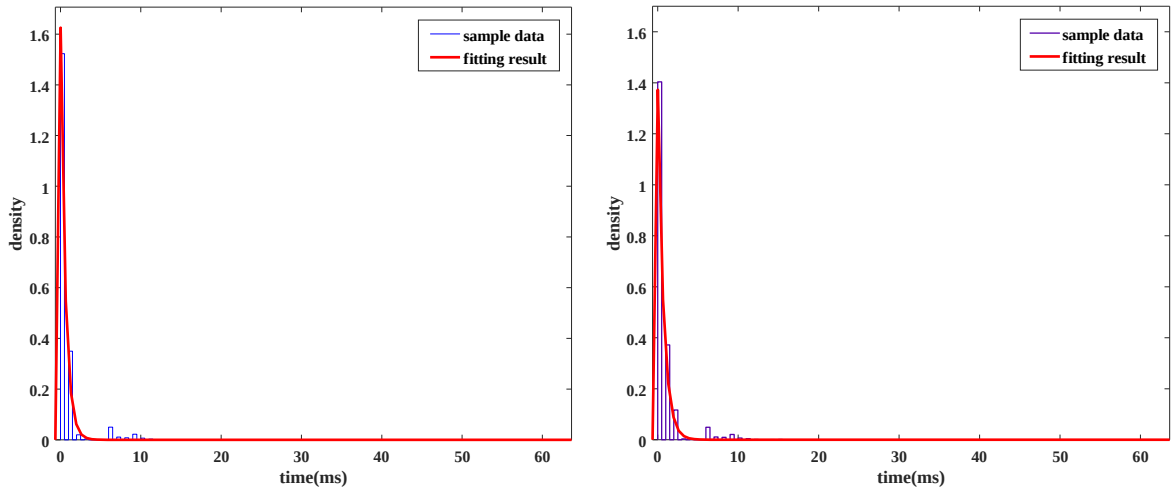


Figure 9. Timeline for supports of X_1 and X_2.



(**a**) Fitting result of Y_1 (**b**) Fitting result of Y_2

Figure 10. Fitting results for sample values of Y_1 and Y_2. (**a**) Fitting result of Y_1; (**b**) Fitting result of Y_2.

Since X_1 and X_2 are independent by the reasonable assumption, Y_1 and Y_2 are also independent. Therefore, for all $\alpha, \beta > 0$, the success probability of recovery is given by

$$\begin{aligned} Pr[Y_1 \leq \alpha, Y_2 \leq \beta] &= Pr[Y_1 \leq \alpha] Pr[Y_2 \leq \beta] \\ &= \int_0^{\alpha} 1.68 e^{-1.68y} dy \int_0^{\alpha} 1.41 e^{-1.41y} dy \\ &= (1 - e^{-1.68\alpha})(1 - e^{-1.41\beta}). \end{aligned} \tag{1}$$

Before we introduce the probabilistic models, we have checked whether the supports and parameter λs depend on hardware factors in the same model. We have uploaded Brillo on three Edison boards and tested each support and distribution on each board. Table 3 shows that the influence of hardware dependency is negligible and supports of our model are convincing.

Table 3. Supports of $cycles_1$ and $cycles_2$ on each Edison board.

Device No.	# 1	# 2	# 3
$supp(X_1)$	[2803, 2867]	[2804, 2984]	[2803, 2813]
$supp(X_2)$	[3513, 3577]	[3515, 3694]	[3513, 3524]

3.3.2. Success Probability of Recovery and Cost of Attack

From Equation (1), Table 4 indicates several trade-offs between success probabilities and costs of the attack. We define the cost of attackers by the number of operations (operating the extract function) for an attacker to obtain and confirm a correct random number. In fact, the values of *cycles* vary by nanosecond resolution on the Edison device. However, an attacker can observe only millisecond precision on the outside, and then the attacker has to guess the nanosecond out of 10^6, equivalent to 2^{20}.

For example, if the attacker observes a generated random number with values of $cycles_1 = 1$ (ms) and $cycles_2 = 2$ (ms) in the $interval_3$ (e.g., $output_{20}$). Then, he will guess correct $cycles_1$ and $cycles_2$ by generating random numbers from $cycles_1 = 0$ and $cycles_2 = 0$. However, before he guesses the values, he has to set a success probability of recovery and the maximum cost because he does not know how he has to operate *extract unit*. If the attacker selects the values of $\alpha = 2.09$ and $\beta = 2.49$ from $Pr[Y_1 \leq \alpha] = 0.95$ and $Pr[Y_2 \leq \beta] = 0.95$, he can obtain the success probability of 0.90 ($= 0.95 \times 0.95$). On the other hand, he needs the maximum cost of 5.20×2^{40} ($= 2.09 \cdot 2^{20} \times 2.49 \cdot 2^{20}$) to search with nanosecond precision. In this case, he finds the correct values of $cycles_1$ and $cycles_2$ with 2.00×2^{40} ($= 1.00 \cdot 2^{20} \times 2.00 \cdot 2^{20}$) trials. In the next section, we simulate this scenario.

Table 4. Several trade-offs between success probability and attack cost.

α	β	$Pr[Y_1 \leq \alpha]$	$Pr[Y_2 \leq \beta]$	$Pr[Y_1 \leq \alpha, Y_2 \leq \beta]$	$\alpha \cdot 2^{20} \times \beta \cdot 2^{20}$
0.83	0.98	0.75	0.75	0.56	0.81×2^{40}
1.31	1.57	0.80	0.80	0.64	2.10×2^{40}
1.50	1.79	0.85	0.85	0.72	2.69×2^{40}
1.78	2.12	0.90	0.90	0.81	3.77×2^{40}
2.09	2.49	0.95	0.95	0.90	5.20×2^{40}
64.00	64.00	1.00	1.00	1.00	$4096 \times 2^{40} (\approx 2^{52})$

4. Experimental Results for Success Probabilities and Costs

In this section, we make a scenario to demonstrate the success probability of recovery. We benchmark a case of an attack in [7]. As a discriminant, the leak of random numbers is necessary to ensure that the random numbers are successfully recovered. Therefore, we assume that an attacker can obtain several random numbers, which are exposed to the outside, at least one time. In order to compare the leak value with the guessing value, the attacker performs an exhaustive search using the *extract unit*. If the attacker finds the right value, he can obtain the state of the non-blocking pool. In the other words, he obtains the right value of $cycles_1$ and $cycles_2$. From these values, he also recovers all of random numbers in $inverval_1$, $interval_2$, and $interval_3$.

Algorithm 1 represents the procedure to recover the random numbers. This algorithm receives some input parameters: a leak random number as a discriminant (d), success probabilities (p_1, p_2), parameters of exponential distributions (λ_1, λ_2), starting points of searching space (s_1, s_2), and an index of the leak random number (i). From these parameters, Algorithm 1 returns several results: values of *cycles* (c_1, c_2), an internal state of the non-blocking pool, and an array of the random sequence in the *intervals* ($r[]$), where $r[i]$ indicates i-th output.

Algorithm 1: Recovery of random numbers from an output leak.

Input: $d, p_1, p_2, \lambda_1, \lambda_2, s_1, s_2, i$
Output: $(c_1, c_2), state, r[]$ or no solution

$\alpha \leftarrow -\frac{1}{\lambda_1}\ln(1-p_1) \times 2^{20}$ ▷ $Pr[X_1 \leq \alpha] = p_1 = 1 - e^{-\lambda_1 \alpha}$
$\beta \leftarrow -\frac{1}{\lambda_2}\ln(1-p_2) \times 2^{20}$ ▷ $Pr[X_2 \leq \beta] = p_2 = 1 - e^{-\lambda_2 \beta}$
for $c_1 = s_1$ **to** $s_1 + \alpha$ **do**
  for $c_2 = s_2$ **to** $s_2 + \beta$ **do**
    `initialize_entropy_pool(`$state$`)`
    $r[] \leftarrow$ `extract_unit(`$c_1, c_2, state$`)`
    if $d == r[i]$ **then**
      return $c_1, c_2, state, r[]$
  end
end
return no solution

For example, if $ouput_{20}$ is used in a nonce value of ClientHello of SSL, $ouput_{21}$ becomes the PreMasterSecret. It is encrypted with the public key of the server and then transmitted from the client to the server. After this communication, both the client and server generate the shared master key. Because $ouput_{20}$ is exposed in the public channel, an attacker can obtain $ouput_{20}$ and compromise the state of the non-blocking pool by using the leak value as a discriminant. Figure 11 depicts this scenario.

According to this scenario, we implemented a program to recover random numbers of 700 bytes between $interval_1$ and $interval_3$ using $ouput_{20}$ with $\alpha = 2.09$ and $\beta = 2.49$. As the results, it requires 12 h to find the right value with approximate 2^{35} trials of the *extract unit* (SHA-1 $\times$ 2 + mixing function $\times$ 1). It is not a real-time analysis on a single PC. However, the attack program can be drastically accelerated with parallel computing.

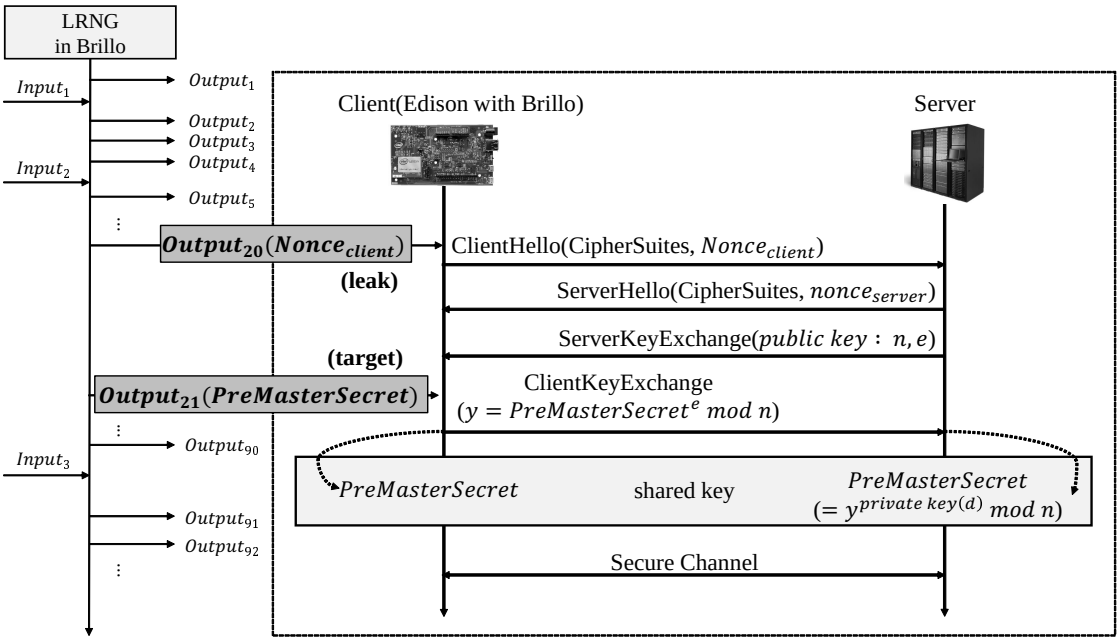


Figure 11. Scenario to recover random numbers during boot time.

5. Several Countermeasures

We determined that two features of LRNG exist in Brillo. Using these features, the random numbers of 700 bytes during boot time of Brillo can be recovered with the probability of 90% by time

complexity for 5.02×2^{40} trials of *extract unit*. Two approaches for mitigating this vulnerability can be considered. One can be immediately applied, and the other requires systematic works.

- Simple methods to apply
  - Reconfiguring the threshold less than 192 bits: the rate of the entropy transfer can be improved to make the randomness of the non-blocking pool active.
  - Reducing the interval size: recoverable random numbers can be larger when the interval length is long. If the interval length is limited, it is possible to minimize recoverable random numbers even though the state of the non-blocking pool is compromised.
- Methods requiring systematic research
  - Establishing a theoretical model of entropy estimation: Because the entropy estimation algorithm of LRNG is conservative, the entropy is underestimated and impacts efficient random number generation. The underlying cause is not apparent in the theory of estimating entropy; therefore, a theoretical analysis is needed.
  - Designing efficient use of the entropy source: LRNG outputs random numbers by 10 bytes. In most cases, the requested size of the random number is less than 10 bytes; the remaining parts are not used. This results in wasted entropy sources. Consequently, a deterministic interval can be long. If multiple entropy pools are designed to manage wasted random numbers, efficient use of entropy sources is possible.

6. Conclusions

LRNG has several security problems while operating on a PC and smartphone. We investigated whether identical problems occur in the Linux-kernel-based operating system, Brillo. We observed two features of LRNG when operating during Brillo boot time. Random numbers of 700 bytes can be recovered with the probability of 0.90 by the cost of 5.20×2^{40}. In conclusion, the entropy of random numbers of 700 bytes is approximately 43 bits. This means that structural improvements and theoretical analysis of its security are required. Various methods can be proposed to improve security.

In the future, we will study LRNG in three perspectives. The first will be to analyze unknown usage of remaining random numbers. The results can be used to find several vulnerabilities for the init process and cryptographic libraries. Secondly, we are planning to consider brand-new boards and their hardware properties later because these boards may expose different features with Edison. Lastly, we will study efficient use of entropy sources in terms of the design. In order to overcome LRNG inefficiency and conservatism, several efficient methods are needed to enable optimal usage of the noise sources.

Acknowledgments: This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Plannig (No. NRF-2016M3C4A7030648).

Author Contributions: Taeill Yoo contributed to: suggestion of this topic, collecting data, experiments, and writing the draft. Ju-Sung Kang contributed to: suggestion of theoretical model and analysis of the data using the probability model. Yongjin Yeom, who is corresponding author, contributed to: leading the research direction, provision of many ideas of analysis, and discussion of the results. All authors have read and approved the final manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Goldberg, I.; Wagner, D. Randomness and the netscape browser. *Dr. Dobb's J.* 1996, *10*, 67–71.
2. Yilek, S.; Rescorla, E.; Shacham, H.; Enright, B.; Savage, S. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference (IMC 2009), Chicago, IL, USA, 4–6 November 2009; Volume 10, pp. 15–27.

3. Heninger, N.; Durumeric, Z.; Wustrow, E.; Halderman, J.A. Mining your ps and qs: Detection of widespread weak keys in network devices. In Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, 8–10 August 2012; pp. 205–220.
4. Bernstein, J.D.; Chang, Y.; Cheng, C.; Chou, L.; Heninger, N.; Lange, T.; van Someren, N. Factoring RSA keys from certified smart cards: Coppersmith in the wild. In Proceedings of the 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, 1–5 December 2013; pp. 341–360.
5. Mowery, K.; Wei, M.; Kohlbrenner, D.; Shacham, H.; Swanson, S. Welcome to the entropics: Boot-time entropy in embedded devices. In Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, 19–22 May 2013; pp. 589–603.
6. Ding, Y.; Peng, Z.; Zhou, Y.; Zhang, C. Android low entropy demystified. In Proceedings of the IEEE International Conference on Communications (ICC), Sydney, Australia, 10–14 June 2014; pp. 659–664.
7. Kim, S.H.; Han, D.; Lee, D.H. Predictability of Android OpenSSL's pseudo random number generator. In Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13), Berlin, Germany, 4–8 November 2013; pp. 659–668.
8. Kaplan, D.; Kedmi, S.; Hay, R.; Dayan, A. Attacking the linux PRNG on android: Weaknesses in seeding of entropic pools and low boot-time entropy. In Proceedings of the 8th USENIX Workshop on Offensive Technologies, San Diego, CA, USA, 19 August 2014.
9. Michaelis, K.; Meyer, M.; Schwenk, J. Randomly failed! The state of randomness in current java implementations. In Proceedings of the Topics in Cryptology—CT-RSA 2013—The Cryptographers' Track at the RSA Conference 2013, San Francisco, CA, USA, 25 February–1 March 2013; pp. 129–144.
10. Bernstein, J.D.; Lange, T.; Niederhagen, R. Dual EC: A standardized back door. In *The New Codebreakers—Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 256–281.
11. Checkoway, S.; Niederhagen, R.; Everspaugh, A.; Green, M.; Lange, T.; Ristenpart, T.; Bernstein, J.D.; Maskiewicz, J.; Shacham, H.; Fredrikson, M.; et al. On the practical exploitability of dual EC in TLS implementations. In Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2014; pp. 319–335.
12. Gutterman, Z.; Pinkas, B.; Reinman, T. Analysis of the linux random number generator. In Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006), Berkeley, CA, USA, 21–24 May 2006; pp. 371–385.
13. Hay, H.; Dayan, A. Android Keystore Stack Buffer Overflow. Available online: https://dl.packetstormsecurity.net/1406-exploits/android-keystore-stack-buffer-overflow.pdf (accessed on 10 March 2017).
14. Google. Available online: https://android.googlesource.com/brillo/manifest/ (accessed on 10 March 2017).
15. Lacharme, P.; Röck, L.; Strubel, V.; Videau, M. The linux pseudorandom number generator revisited. Available online: https://eprint.iacr.org/2012/251.pdf (accessed on 13 March 2017).
16. Android, Brillo, ChromeOS in Slideshare. Available online: https://www.slideshare.net/l_b__/androidbrillochromeos (accessed on 10 March 2017). (In Japanese)
17. Android Interfaces and Architecture. Available online: https://source.android.com/devices/ (accessed on 10 March 2017).
18. Intel. Available online: https://software.intel.com/en-us/iot/hardware/edison (accessed on 10 March 2017).
19. Brillo OS—Successfully Flashing. Available online: https://communities.intel.com/thread/96539 (accessed on 10 March 2017).
20. Grimmett, G.R.; Stirzaker, D.R. *Probability and Random Processes*, 3rd ed.; Oxford University Press: Oxford, UK, 2001.
21. Mathworks, Version R2016b. Available online: https://kr.mathworks.com/help/stats/model-data-using-the-distribution-fitting-tool.html (accessed on 10 March 2017).