2013 IEEE Symposium on Security and Privacy

Welcome to the Entropics: Boot-Time Entropy in Embedded Devices

Keaton Mowery, Michael Wei, David Kohlbrenner, Hovav Shacham, and Steven Swanson
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA

***Abstract*—We present three techniques for extracting entropy during boot on embedded devices.**

Our first technique times the execution of code blocks early in the Linux kernel boot process. It is simple to implement and has a negligible runtime overhead, but, on many of the devices we test, gathers hundreds of bits of entropy.

Our second and third techniques, which run in the bootloader, use hardware features — DRAM decay behavior and PLL locking latency, respectively — and are therefore less portable and less generally applicable, but their behavior is easier to explain based on physically unpredictable processes.

We implement and measure the effectiveness of our techniques on ARM-, MIPS-, and AVR32-based systems-on-a-chip from a variety of vendors.

I. Introduction

Random numbers unpredictable by an adversary are crucial to many computing tasks. But computers are designed to be deterministic, which makes it difficult to generate random numbers. Substantial effort has gone into developing and deploying subsystems that gather and condition entropy, and that use it to generate random numbers on demand.

In this paper, we take an extreme position: Randomness is a fundamental system service; a system cannot be said to have successfully booted unless it is ready to provide high-entropy randomness to applications.

Our main contributions are three techniques for gathering entropy early in the boot process — before interrupts are enabled, before a second kernel thread is spawned. Our techniques are suitable for use even on embedded systems, where entropy-gathering is more challenging than on desktop PCs. We implement our proposed techniques and assess their effectiveness on systems-on-a-chip (SoCs) that integrate ARM, MIPS, and even AVR32 CPU cores.

Motivation: Our work is inspired by the recent paper of Heninger, Durumeric, Wustrow, and Halderman [16], which uncovered serious flaws in the design and implementation of the Linux kernel's randomness subsystem. This subsystem exposes a blocking interface (`/dev/random`) and a nonblocking interface (`/dev/urandom`); in practice, nearly all software uses the nonblocking interface. Heninger et al. observe (1) that entropy gathered by the system is not made available to the nonblocking interface until Linux estimates that 192 bits of entropy have been gathered, and (2) that Linux is unnecessarily conservative in estimating the entropy in events, and in particular that on embedded systems no observed events are credited with entropy. These two facts combine to create a "boot-time entropy hole," during which the output of `/dev/urandom` is predictable.

The Linux maintainers overhauled the randomness subsystem in response to Heninger et al.'s paper. The timing of every IRQ is now an entropy source, not just IRQs for hard disks, keyboards, and mice. Entropy is first applied to the nonblocking pool, in the hope of supplying randomness to clients soon after boot. (Clients waiting on the blocking interface can block a bit longer.)

The new design leaves in place the race condition between entropy accumulation and the reading of supposedly random bytes from the nonblocking pool. It would be better, we argue, to gather entropy so early in the boot process that all requests for randomness can be satisfied.

In this paper, we present entropy-gathering techniques that realize this vision. We show how to gather entropy in the bootloader or early in the kernel boot process on embedded systems running a variety of popular processors. Our techniques require neither the multicore x86 processor of desktop PCs nor the sophisticated sensors available to smartphones. They do not require network connectivity. They can be used in place of, or side by side with, Linux's current entropy-gathering infrastructure.

Our three techniques provide different tradeoffs along three metrics: (1) How many random bits can be obtained, and how quickly? (2) How much system-specific knowledge is required to implement the technique? (3) To what extent can the entropy obtained be explained by well-studied physical processes that are believed to be unpredictable? None of our proposed techniques is ideal along all three metrics.

Our first technique: Instruction timing early in kernel boot: In our first technique, we instrument the kernel's startup code to record how long each block of code takes to execute. This approach has previously been used to gather entropy in userland code; we show that it is also applicable when a single kernel thread of execution runs, with interrupts disabled, on an embedded system. On many of the devices we tested (see Section II), this technique gathers a surprisingly large amount of entropy — over 200 bits on the Raspberry Pi, for example — at negligible runtime overhead; on other devices, less entropy is available.

We have not been able to account conclusively for the large amount of entropy this technique gathers on some

DOI 10.1109/SP.2013.46

devices or for the smaller amount it gathers on other devices. In Section III, we pinpoint architectural features that are partly responsible.

Our second and third techniques: DRAM decay and PLL locking: In our second class of techniques, we take advantage of architectural features that vary between SoCs, rendering them less portable and less widely applicable, but promising more entropy. In addition, we are able to pinpoint more precisely the sources of the entropy we measure.

In Section IV, we show that it is possible for bootloader code, running from on-chip SRAM, to turn off DRAM refresh. With refresh disabled, the contents of DRAM decay unpredictably; we exploit this fact to obtain an entropy source. In Section V, we show that our ability to repeatedly reconfigure a peripheral clock on the BeagleBoard xM translates into another high-rate entropy source.

A. Related Work

As noted above, the motivation for our paper is Heninger et al.'s recent study of the Linux randomness subsystem [16].

Random number generation is hard, and flaws in randomness subsystems have been identified with dismaying regularity. In 1996, Goldberg and Wagner analyzed the random number generator in the Netscape browser [10]. A decade later, Luciano Bello found that the OpenSSL package shipped with Debian and Ubuntu had a broken random number generator [37]. The bug's effects were quantified by Yilek et al. [41]. Cryptographers have designed "hedged" cryptosystems whose security degrades as little as possible in the absence of good randomness [2]. Otherwise secure random number generators can break in novel settings: Ristenpart and Yilek observed that virtual machine resets could lead to randomness reuse and proposed solutions [31, 40].

Researchers have expended considerable effort considering how best to design randomness subsystems. Gutmann described design principles for random number generators [11]; Kelsey, Schneier, Wagner, and Hall proposed a formal security model for random number generators and described attacks on deployed systems [23]. Kelsey, Schneier, and Ferguson then proposed Yarrow, a concrete design for a family of random number generators [24]. More recently, NIST has made recommendations for producing random numbers from an entropy pool [1]. Researchers have also studied the effectiveness of the randomness subsystems deployed with Linux [12, 26] and Windows [7]. Gutterman, Pinkas, and Reinman, in their study of Linux randomness system [12] specifically pointed out the vulnerability of Linux-based routers like those running OpenWRT software.

Entropy can be obtained from many sources: from dedicated hardware, using analog feedback circuits such as phase-locked loops (PLLs) [9] or digital feedback circuits (as included in Intel's latest processors [4, 14]); from timing other hardware devices, such as hard disks [6, 20]; from timing user input; or, in sensor-rich devices such as smartphones, from sensor noise in microphones [8, Section 5.3.1], cameras [3], and accelerometers [38].

Instruction timings have long been used as a source of entropy. In Section II-A we describe Bernstein's `dnscache-conf` program from 2000. The method was explored in detail in the HAVENGE system of Seznec and Sendrier [33]. In both cases, the entropy is assumed to derive from the unpredictable arrival times of interrupts and the behavior of the system scheduler. By contrast, our first technique (described in Section II) obtains entropy even with interrupts disabled and a single thread of execution.

Pyo, Pae, and Lee, in a short note, observe that DRAM refresh timings are unpredictable, which means that DRAM access timings can be used as an entropy source [30].

Theoretical grounding for the unpredictability of instruction timing was given by McGuire, Okech and Zhou [27] and Mytkowicz, Diwan, and Bradley [28]. These papers consider x86 chips; the processors we study are considerably simpler.

Decay patterns in RAM, used in our second technique (described in Section IV), have also been considered before. Holcomb, Burleson, and Fu use SRAM decay as an entropy source on RFID devices [18]. Halderman et al. studied DRAM decay patterns in detail [13].

II. Early Kernel Entropy

Our first method for gathering entropy is an application of a simple idea: After each unit of work in a code module, record the current time using a high-resolution clock. Specifically, we instrument `start_kernel`, the first C function run in the Linux kernel on boot, and use the cycle counter as our clock.

Our approach is attractive. It runs as early as possible in the kernel boot process: All but one use of randomness in the Linux kernel occurs after `start_kernel` has completed. It imposes almost no performance penalty, requiring, in our prototype implementation, 3 KiB of kernel memory and executing a few hundred assembly instructions. It is simple, self-contained, and easily ported to new architectures and SoCs.

The question is, Does it work? Previous applications of the same idea ran in user mode on general-purpose x86 machines. They could take advantage of the complexity of the x86, the unpredictable arrival timing of interrupts, interleaved execution of other tasks, and the overhead of system call servicing when accessing a high-resolution clock. By contrast, our code runs on an embedded device with interrupts disabled and a single thread of execution. Nevertheless, we are able to extract a surprising amount of entropy — in some cases, hundreds of bits.

In this section, we discuss our implementation and evaluate its effectiveness on ARM SoCs from six vendors, a MIPS SoC, and an AVR32 SoC. In Section III, we discuss architectural mechanisms that are partly responsible for the entropy we observe.

A. Genesis

In 2000, Daniel J. Bernstein released `dnscache` 1.00, a caching DNS recursive resolver that is now part of the `djbdns` package. DNS resolvers generally operate over UDP, which means that an interested attacker can spoof the answer to a query by simply forging a packet. To combat this, each DNS query carries along with it a pre–selected port number and query ID, which the response must have to be considered valid. Therefore, `dnscache`, when acting as a client of other DNS servers, must be able to choose these port numbers and query IDs well [19, 22].

One of `dnscache-conf`'s duties is to provide entropy that will later be used by `dnscache`. To gather this entropy, the `dnscache-conf` utility simply instruments its own startup procedure with multiple calls to `gettimeofday()`, and mixes each result into the entropy pool. Due to the cost of each syscall, unpredictable hardware interrupts, OS process scheduling, clock skew, and a host of other factors, this method provides `dnscache-conf` with high-quality entropy for the cost of a few extra syscalls. An excerpt from `dnscache-conf.c`:

```
makedir("log");
seed_addtime();
perm(02755);
seed_addtime();
makedir("log/main");
seed_addtime();
owner(pw->pw_uid,pw->pw_gid);
seed_addtime();
perm(02755);
seed_addtime();
```

A method which works in userland on an x86 machine might not apply to kernel-level code on much simpler embedded devices. Indeed, we were initially skeptical: In the absence of interrupts, multiple threads, syscall overhead, and on simpler processors than the x86, would there still be enough variation to make such a scheme viable?

B. Methodology

1) Kernel Instrumentation: To collect information about the kernel boot process, we modified a Linux kernel for each system we examined. Our kernel instrumentation consists of a basic macro that can be inserted anywhere in kernel boot to record the current cycle count with low overhead. The macro recorded the current cycle count to an incrementing index in a statically allocated array. We incremented the index at compile time, and thus the only operations performed by the measurement at run time are reading the cycle counter and a single memory store.

We inserted the macro between every function call in `start_kernel`, the first C function called during kernel boot. The majority of the code executed during this sequence is straight-line, with a varying number of instructions executed during each function call. We chose this sampling method because it offered the simplest patch to the kernel at the earliest point in the boot process. Our instrumentation then printed the measured times to the kernel log. An init script copied out the relevant data from the log, truncated the log, and immediately restarted the system using `reboot`. Temperature data was not collected. In this manner, we gathered data on thousands of restarts per day per machine with minimal interaction. Machines were switched off and the data pulled after 24–48 hours of continuous rebooting and data collection.

To estimate the performance overhead, we implemented a "production-ready" version, which skips printing to the kernel log in lieu of mixing the results directly into the kernel's randomness pools. We then used the cycle counter to measure the execution time of `start_kernel`, both with and without our instrumentation. On the Raspberry Pi (detailed in Section II-C3), our technique adds approximately 0.00019 seconds to the kernel boot process.

2) Devices: As described in the previous section, we instrumented a variety of Linux kernels and ran them on a broad variety of embedded platforms, ranging from high-powered ARM computers to low-end special-purpose MIPS and AVR devices.

ARM: ARM, Inc. licenses its processor architecture to many companies that integrate ARM cores into their designs. Two systems-on-a-chip that integrate the same ARM core might nevertheless have very different performance characteristics. To check the general applicability of our approach to ARM-based embedded systems, we instrumented and collected data from systems-on-a-chip from many of the most prominent ARM licensees: Broadcom, Marvell, NVIDIA, Texas Instruments, Qualcomm, and Samsung. These vendors represent six of the top seven suppliers of smartphone processors by revenue.

Specifically, the first system we tested was the Raspberry Pi, which contains a Broadcom BCM2835 SoC featuring a 1176JZF-S core, which is an ARM11 core implementing the ARMv6 architecture. We also instrumented the BeagleBoard xM, which uses a Texas Instruments DM3730 containing a ARMv7 Cortex-A8; the Trim-Slice featuring an NVIDIA Tegra 2, a ARMv7 Cortex-A9; the Intrinsyc DragonBoard, with a Qualcomm SnapDragon SoC containing a Qualcomm Krait ARMv7; the FriendlyARM Mini6410 with a Samsung S3C6410, another version of the ARM1176JZF-S ARM11 ARMv6 core; and the Cubox, which uses a Marvell ARMADA 510 SoC containing a Sheeva ARMv7 core.

MIPS: Previous work on embedded device entropy identified routers as important targets, as they are conveniently located to inspect and modify network traffic and, as reported by Heninger et al. [16], routinely ship with extremely poor entropy, as evidenced by their SSL certificates.

With this in mind, we instrumented the early Linux boot process on the Linksys WRT54GL router, containing a Broadcom 5352EKPBG 200MHz MIPS "router-on-a-chip." Revered for their extensibility, the WRT54GL represents a basic wireless router as found in the homes of millions.

AVR32: Finally, we instrumented a kernel for the Atmel NGW100 mkII, which contains a AT32AP7000-U AVR32 core. The AVR32, designed by Atmel, represents one of the smallest and lowest-power CPUs capable of running Linux. Even on the AVR32, our techniques uncover substantial randomness. The existence of instruction entropy on this platform indicates that execution randomness is not solely due to processor optimizations and complexity.

C. Results and Analysis

In this section, we will discuss the results of each device's kernel instrumentation, and the expected quality of the entropy extracted.

As the existence of true randomness is an open philosophical question (and therefore beyond the scope of this paper), we will treat entropy as "unpredictability": given the knowledge that a remote attacker can possibly have, how difficult would it be to guess the device–generated random bits?

1) Statistical Tests: We are unable to conclusively pinpoint and characterize every source of entropy in these systems. Therefore, this analysis will deal only with empirical measurements, as sampled from each board over many boots. We will rely mainly on two estimations: distribution entropy and min-entropy.

Distribution entropy represents, for a given empirical sample, the Shannon entropy of the underlying distribution. For example, a set of samples consisting of 50 A's and 50 B's would have a single bit of distribution entropy, while a set of samples consisting of 1024 unique values has a distribution entropy of 10 bits. Distribution entropy can be calculated, for a set S of n distinct observed values V_i, each being seen C_i times, with $C = |S| = \sum_{i=0}^{n}(C_i)$, as:

$$D(S) = -\sum_{i=1}^{n} \frac{C_i}{C} \cdot \lg\left(\frac{C_i}{C}\right) \tag{1}$$

Note that distribution entropy will almost always underestimate the entropy of the underlying distribution. That is, the distribution entropy calculated from a empirical sampling S will always be less than or equal to $\lg(|S|)$, regardless of the actual entropy of the underlying distribution.

Our other empirical estimator, min-entropy, measures the prevalence of the most common element in a distribution. In other words, if an adversary is allowed a single guess at the value, min-entropy measures how often she will be correct. For a set S of n distinct observed values V_i with counts C_i, the min-entropy is:

$$M(S) = -\lg\left(\frac{max_i(C_i)}{C}\right) \tag{2}$$

With these two metrics, we can characterize the distributions sampled from each device and predict their real-world entropy content.

2) Entropy Extraction: Furthermore, each boot sequence generates a vector of test times, one per test. In our analysis, we will examine both the sampled distributions of individual test times, as well as the sampled distribution of test vectors. The test vector, once generated, can be fed into an entropy extractor to produce an evenly–distributed random seed, which can then used to seed kernel pseudo-random number generators.

The values in the test vector are partly correlated: if nothing else, later tests have cycle counts larger than earlier tests. Extracting the entropy from such a source is a challenging theoretical problem [29], but under the random oracle heuristic simply applying a cryptographic hash to the test vector is sufficient. NIST has published explicit recommendations for implementing what they call "reseeding" in randomness generators [1].

3) Raspberry Pi: The Raspberry Pi is a popular single-board ARM computer, built around the Broadcom BCM2835 System–on–a–chip (SoC), which contains an ARM 1176JZF-S ARM11 ARMv6 core clocked at 700 MHz. We modified the Linux 3.2.27 kernel provided for the Raspberry Pi[1] to perform our data collection. This involved enabling and configuring the hardware cycle counter and the two hardware performance counters present in the ARM 1176JZF-S, as well as surrounding each function call in `start_kernel` with instrumentation to record the current counter values, and a final function to dump our results to the kernel log. We were able to surround every function in `start_kernel`, for a total of 78 individual tests.

Next, we booted the instrumented kernel on four identical Raspberry Pis, and recorded the counters for each boot.

In short, almost every test shows a surprising amount of variation in the number of cycles it takes to execute. Figures 1, 2, 3, and 4 show a histogram of test times, in cycles, for tests 4, 5, 36, and 41 as seen across 301,647 boots across all four Raspberry Pi devices. The lighter regions are the contribution of device #0, which, by itself, contributes 130,961 boots. These four graphs are representative of the four classes of histogram that we see on the Raspberry Pi: a "two-normal" distribution like Test 4, a "quantized" distribution like Test 5, a "bimodal plus noise" distribution like Test 36, and a "normal" distribution like Test 41.

For comparison, Test 4 corresponds to the initialization function `cgroup_init_early()`, which is responsible for setting up process groups for resource management, and mostly involves setting memory locations to initial values. Test 5 is `local_irq_disable()`, which disables interrupts. It consists solely of the ARM instruction "`cpsid i`", and the variation in this test is likely due to hardware

[1]Online: https://github.com/raspberrypi/linux

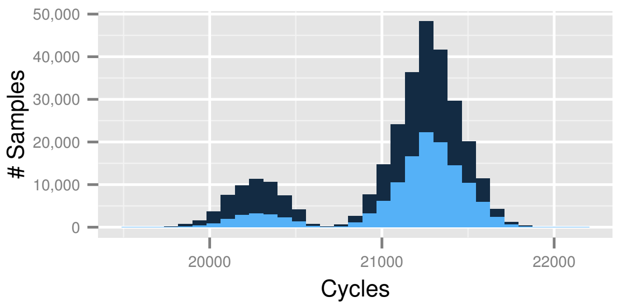

Figure 1: Histogram of cycle counts for Test 4 on 4 Raspberry Pis. Lighter region is data from device #0 only.

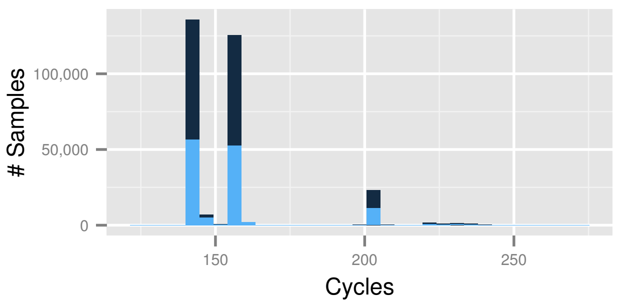

Figure 2: Histogram of cycle counts for Test 5 on 4 Raspberry Pis. Lighter region is data from device #0 only.

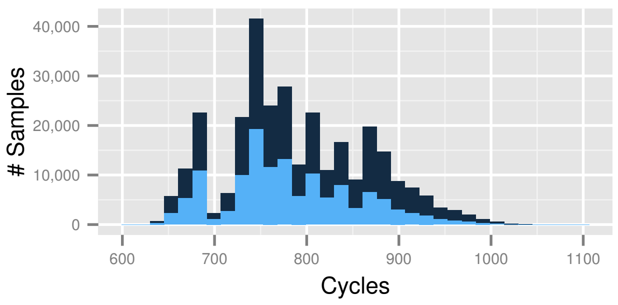

Figure 3: Histogram of cycle counts for Test 36 on 4 Raspberry Pis. Lighter region is data from device #0 only.

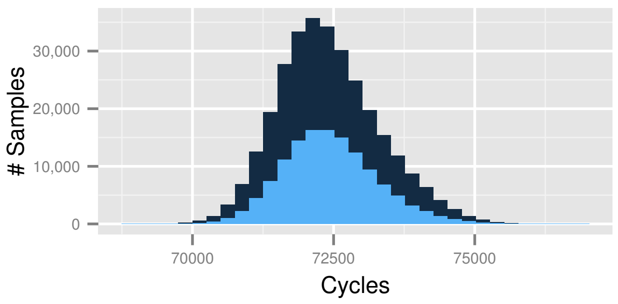

Figure 4: Histogram of cycle counts for Test 41 on 4 Raspberry Pis. Lighter region is data from device #0 only.

initialization state. Test 36 is `prio_tree_init()`, and is simply a small loop which initializes an array. The relatively quantized period of this function is likely due to stalls in memory fetches and stores. Also, note that IRQs remain disabled until Test 45, and so interrupts cannot be blamed for any variation in these test times.

Overall, these distributions are far wider than we initially expected. Test 41, in particular, has a minimum value of 69,098 cycles and a maximum of 76,625, almost 10.9% more. In this region of 7,527 cycles, the data set contains 5,667 distinct test values.

Taken individually, the results of each test give an empirical distribution over the cycles elapsed during execution. If we treat a test as a random variable, we can extract that entropy and use it to seed a random number generator.

To estimate the entropy contribution of each test, we apply the distribution entropy calculation to our observed data. The results of this calculation are in Table I.

However, further investigation is needed before we proclaim success. While each test has between 0.45 and 12.99 bits of distribution entropy, we cannot naively sum these numbers and proclaim that to be our total entropy produced. In order for that approach to be valid, each test must be statistically independent — the time taken for test T must not depend on the results for tests $(0,\ldots,T-1)$. If, in the worst case, T was a known function of $(0,\ldots,T-1)$, then it would not contribute any entropy whatsoever to the overall total, even if it had plenty of distribution entropy: Its distribution entropy would already be counted by the preceding tests. (Note that we can always safely mix the results of T into the entropy pool. In the worst case, doing so adds no benefit.)

We applied a straightforward correlation test to the data we gathered from the Raspberry Pi and our other systems. More sophisticated tests are possible, for example using NIST's test suite [32]. Specifically, we computed the correlation coefficients between each pair of tests. We can then select a threshold of acceptable risk, and exclude from our entropy estimate any tests which are correlated with another test beyond that limit. Figure 5 shows the full entropy estimate function for the Raspberry Pi for all possible thresholds. This function is surprisingly linear, suggesting that the Raspberry Pi tests, while correlated, do not cluster together when taken as a whole. With no self-evident threshold to pick, we arbitrarily exclude from our entropy estimate any tests having correlation of 0.4 or more with another test.

Applying this test to our Raspberry Pi data, we find some intriguing results. Figure 6 shows a scatterplot of Test 4 and Test 7. The former is the kernel function `cgroup_init_early`, which is responsible for initializing resource–managing process groups, and mainly consists of initializing variables throughout kernel memory. The latter, on the other hand, is `boot_cpu_init`, which is in charge of marking the CPU as "online" and "active", allow-

Test	RPi	BB	Trim-Slice	Dragon Board	Mini-6410	Cubox	WRT	NGW
#0	4.07	4.10	5.70	9.48	-	0.12	8.14	4.33
#1	1.91	7.31	4.31	4.30	0.55	5.02	6.82	0.-
#2	0.85	2.32	4.77	2.33	0.-	0.-	7.25	1.80
#3	8.51	2.58	9.97	10.34	0.29	0.42	4.66	0.-
#4	9.32	8.76	10.88	9.51	5.41	0.-	4.66	0.-
#5	2.58	2.78	2.95	2.74	0.17	0.-	4.15	0.77
#6	6.58	2.33	8.86	5.04	0.51	1.21	8.04	2.36
#7	5.25	5.45	7.66	5.44	3.28	2.12	8.80	1.05
#8	2.24	3.85	9.45	8.77	0.34	2.45	2.21	1.80
#9	8.79	5.09	10.98	8.97	6.03	4.55	3.57	2.19
#10	13.42	9.32	11.28	14.21	7.74	7.92	6.60	2.19
#11	2.73	11.42	8.54	4.01	1.78	0.-	7.87	2.19
#12	2.73	6.25	7.97	2.47	6.34	0.-	7.99	2.19
#13	8.48	5.27	9.77	8.63	0.03	1.68	8.70	1.00
#14	2.28	9.82	8.67	6.06	7.47	0.-	0.78	1.58
#15	12.19	7.54	10.79	10.85	0.79	2.41	0.41	1.00
#16	2.00	11.12	7.95	4.96	6.93	0.-	1.59	0.-
#17	9.83	6.40	9.95	10.20	0.70	1.52	6.22	0.-
#18	3.88	10.04	8.38	8.44	6.95	0.-	9.92	1.00
#19	8.48	8.27	9.40	7.67	7.30	0.01	9.09	0.-
#20	11.43	9.21	10.43	6.65	7.32	2.53	8.66	1.00
#21	3.24	6.15	3.80	11.12	6.74	0.-	7.28	1.00
#22	6.30	10.59	8.59	6.37	1.27	1.05	8.66	0.-
#23	10.85	6.63	10.18	9.72	7.60	1.50	9.34	1.00
#24	13.52	5.89	10.51	9.68	6.82	1.14	-	1.00
#25	9.85	10.55	10.28	10.83	4.94	1.84	-	0.-
#26	2.29	10.17	6.26	4.05	7.73	0.-	-	1.00
#27	14.15	8.47	11.29	14.43	7.73	2.90	-	1.00
#28	8.14	4.85	10.41	10.24	4.07	1.77	-	1.00
#29	3.80	11.77	8.68	4.97	7.36	0.-	-	0.-
#30	9.60	10.20	10.01	2.21	7.22	1.99	-	0.-
#31	10.29	3.74	10.03	10.15	7.56	2.07	-	0.-
#32	5.14	3.20	10.05	9.73	7.57	0.-	-	0.-
#33	9.28	9.89	9.54	10.27	4.79	2.18	-	5.90
#34	11.83	9.52	11.02	8.24	6.33	2.78	-	0.02
#35	11.07	10.49	10.76	12.70	4.94	3.57	-	0.-
#36	7.67	9.81	7.94	11.00	4.67	1.12	-	0.-
#37	9.21	10.41	10.29	4.96	7.07	2.12	-	0.-
#38	7.53	11.31	9.25	8.45	7.53	1.26	-	0.-
#39	7.38	9.24	8.68	4.43	0.64	0.10	-	0.14
#40	8.15	8.07	11.17	6.28	7.53	2.96	-	0.12
#41	11.67	7.36	10.81	8.78	4.50	5.49	-	5.07
#42	7.02	8.88	8.41	13.75	1.35	1.04	-	4.27
#43	2.21	11.55	9.24	3.55	9.74	0.-	-	1.25
#44	3.88	7.99	6.43	6.65	4.20	0.93	-	1.90
#45	4.27	9.08	6.70	1.06	0.82	0.-	-	0.-
#46	10.97	4.33	9.05	1.62	2.82	5.57	-	6.00
#47	11.36	3.22	10.67	9.65	0.65	3.85	-	5.35
#48	2.47	4.00	0.06	2.79	1.05	0.-	-	3.55
#49	2.46	6.24	3.48	2.89	0.92	0.-	-	0.-
#50	2.64	11.24	8.03	10.36	0.70	0.08	-	2.28
#51	2.64	7.09	5.91	1.96	8.14	0.-	-	2.02
#52	2.58	5.47	7.08	1.49	2.34	0.-	-	3.01
#53	1.79	5.25	6.97	5.16	9.93	0.-	-	2.80
#54	3.19	6.23	7.38	3.03	9.96	2.75	-	0.-
#55	10.68	4.88	10.12	3.62	12.88	0.08	-	0.-
#56	4.65	4.60	3.31	1.91	11.21	0.02	-	0.-
#57	5.16	4.26	7.99	1.78	10.48	5.37	-	3.83
#58	6.65	10.05	5.11	8.85	3.09	5.06	-	3.25
#59	12.56	5.37	10.91	6.19	8.04	3.51	-	2.52
#60	11.23	5.44	10.33	2.50	9.09	1.68	-	3.25
#61	9.62	5.81	10.09	2.01	10.23	3.15	-	0.-
#62	4.60	11.76	8.93	11.67	8.88	2.83	-	0.-
#63	8.05	10.61	9.10	9.53	0.88	4.03	-	0.-
#64	8.92	10.44	9.97	9.50	0.54	2.03	-	0.-
#65	10.42	5.80	9.85	3.60	1.04	0.06	-	1.28
#66	8.79	8.17	9.22	8.20	11.73	0.-	-	0.-
#67	8.55	8.97	9.41	9.80	9.63	0.-	-	0.-
#68	1.73	9.86	10.71	9.94	6.87	4.15	-	0.-
#69	11.03	9.40	10.45	9.07	11.69	3.70	-	-
#70	12.99	8.20	10.99	7.87	11.20	2.06	-	-
#71	10.42	10.76	10.02	3.20	1.99	4.00	-	-
#72	8.50	10.52	9.52	2.46	0.99	0.-	-	-
#73	10.92	11.15	10.62	12.03	0.77	0.-	-	-
#74	11.79	9.75	10.70	9.79	12.28	0.68	-	-
#75	2.45	9.57	8.82	8.94	1.63	0.-	-	-
#76	9.48	11.23	9.93	10.73	1.22	4.71	-	-
#77	9.45	-	9.70	11.07	1.26	0.-	-	-
Sum	564.98	594.66	683.40	557.84	394.78	129.15	151.41	90.21

Table I: Per-Test Distribution Entropy, in bits

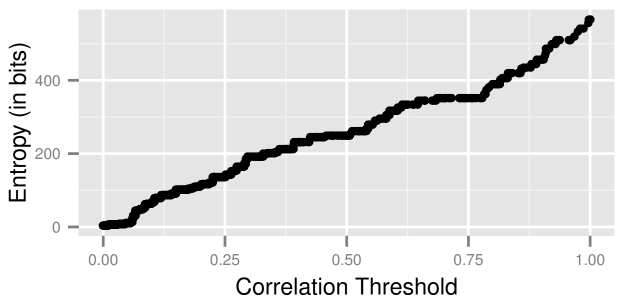

Figure 5: Total Raspberry Pi entropy estimate as a function of acceptable correlation threshold

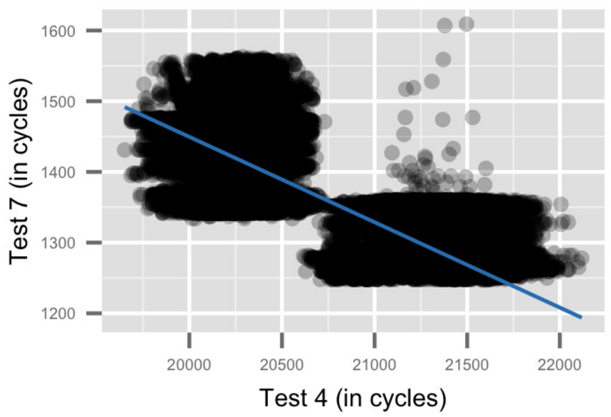

Figure 6: Cycle counts for Tests 4 and 7 on the Raspberry Pi. Correlation coef. = −0.79. Line is the best-fit linear model.

ing other cores to communicate. (Note that the Raspberry Pi has only a single core, but still executes this step.) We have so far been unable to determine a causal relationship between these two tests that might account for the extremely odd relationship in Figure 6.

While we do not believe that the correlations between tests are particularly helpful to an attacker (since a remote or local but post-boot attacker will not have access to the preceding $T-1$ test values), in the interests of caution, we modify our entropy estimate as follows: for each successive variable, add its distribution entropy to the total if and only if, when correlated with each preceding variable in turn, never has a correlation coefficient with magnitude ≥ 0.4. If the variable is thus correlated with a preceding variable, we ignore its sampled distribution entropy entirely.

When computed across the entire Raspberry Pi data set, this conservative estimate places the summed distribution entropy of pairwise uncorrelated variables at 231.9 bits — far beyond the reach of exhaustive-search attacks.

Finally, to ensure that this analysis is not completely off, we compute the distribution entropy over the entire data set of 79-element vectors. For the 301,647 Raspberry Pi boot measurements in our data set, every single one is unique, giving a distribution entropy of 18.2 bits. Since distribution entropy cannot extrapolate beyond the size of the empirical data set, this is an empirical lower bound on the entropy available by simply instrumenting the boot procedure of Linux on the Raspberry Pi, and, given our calculations above, we believe that there is more than sufficient entropy available during the Raspberry Pi's boot process to securely seed the Linux randomness generator.

4) BeagleBoard xM: The BeagleBoard xM is powered by a Texas Instruments DM3730 SoC, containing a 1 GHz Cortex-A8 ARMv7 superscalar CPU core. We acquired and modified a patched Linux 3.2.28-x14[2] to include 77 tests in `start_kernel`.

We have less Linux boot data for the BeagleBoard than our other systems, as we re-purposed the BeagleBoard for other experiments, detailed in Section IV. Nevertheless, we collected data on 3,580 boots.

Per-test distribution entropies for the BeagleBoard are in Table I. Naively summing, these 77 tests give 594.66 bits of entropy between them. Our correlation coefficient threshold test reduces this slightly, to 430.06 bits. As for empirical distribution entropy, all 3,580 boot sequences are unique, giving a distribution entropy floor of 11.81 bits.

5) Trim-Slice: The Trim-Slice is another ARM single-board computer, designed for use as a desktop PC. It contains a 1 GHz NVIDIA Tegra 2 dual-core Cortex-A9 ARMv7 CPU, and a variety of storage options. To stay consistent with our other devices, we chose to boot the Trim-Slice from its MicroSD slot. We modified a Linux 3.1.10-l4t.r15.02 kernel[3] to include our instrumentation, and set the machine to rebooting. Our particular model had an issue of failing to reboot every so often, limiting our data collection for this device.

Nevertheless, we instrumented 2,522 reboots of the Trim-Slice, collecting cycle counts for 78 tests, similar to the Raspberry Pi kernel. Per-test distribution entropy can be found in Table I, giving a total sum of 683.40 bits (which, again, may not be an accurate total estimate). Interestingly, even though the Trim-Slice data set contains 100 times fewer boots than the Raspberry Pi data, the per-test distribution entropies are roughly similar across the board. Since distribution entropy chronically underestimates the entropy of the underlying distribution, this implies that the Trim-Slice's Tegra 2 has a much wider test variation than the ARM 1176JZF-S, which is eminently plausible given that the Tegra 2 is a dual-core platform and based on a Cortex-A9, a larger and more complex core than in the Raspberry Pi.

The Trim-Slice tests also appear to show much less correlation than the Raspberry Pi. When we apply our method of summing only the distribution entropy of variables which are not pairwise correlated with any previous test (cor. coef. ≤ 0.4), the Trim-Slice tests still show a shocking 641.48 bits of entropy. Even if this overstates the actual amount by a factor of 3, there is easily enough entropy extractable on boot to seed any pseudorandom generator.

Finally, as one might expect given the data thus far, each of the 2,522 78-element test vectors sampled on a given Trim-Slice boot is unique, giving a total distribution entropy of 11.30 bits. Again, this represents an empirical lower bound, and is one which we believe is extremely low.

6) Intrinsyc DragonBoard: The Intrinsyc DragonBoard is a fully-featured mobile device development board based around the Qualcomm SnapDragon S4 Plus APQ8060A SoC, which includes a Qualcomm Krait ARMv7 dual-core CPU. Designed as a development board for Android mobile devices, it includes hardware such as a touch screen, wi-fi radio, and a camera module.

As a mobile device development platform, the DragonBoard runs Android 4.0.4 and is backed by a Intrinsyc-modified Linux 3.0.21 kernel. As a result, our patch set was easy to apply. As usual, we inserted 78 tests into `start_kernel`. Instead of a Linux init script for collecting the data, we used the Android `adb` tool to connect to the device via USB, dump the kernel logs and reboot the device. In this way, we collected data on 27,421 boots.

In general, we see excellent entropy generation when booting Linux on the Krait. The per-test distribution entropies can be found in Figure I, with a per-test sum of 557.84 bits. As with our preceeding three ARM SoCs, each boot sequence is unique, giving a empirical distribution entropy of 14.74 bits. The tests are also highly uncorrelated: applying our correlation coefficient threshold test lowers the entropy estimate only slightly to 523.55 bits.

Resource-rich embedded devices, such as phones, have a plethora of available sources of entropy – for example, simply turning on one of their radios. This test, though, shows that our entropy generation technique can protect these devices as well.

7) FriendlyARM Mini6410: The FriendlyARM Mini6410 is yet another single-board ARM device. This particular unit is powered by a Samsung S3C6410A SoC, and contains a ARM 1176JZF-S ARM11 core clocked at 533 MHz. As before, we modified the Linux 2.6.38 manufacturer-provided kernel to instrument `start_kernel`, and inserted 77 tests.

Next, we let the FriendlyARM reboot 46,313 times. Interestingly, the data from the FriendlyARM differs significantly from our other ARM results.

First, the per-test distribution entropies for the FriendlyARM can be found in Table I. (The FriendlyARM tests are offset by one to align identical kernel initialization functions between devices as much as possible.) At first glance, the per-test distribution entropies seem reasonable, given that they are bounded above by $\lg(46,313) = 15.4$ bits, naively summing to 394 bits of entropy.

[2]Online: https://github.com/RobertCNelson/stable-kernel

[3]Online: https://gitorious.org/trimslice-kernel

The oddness arrives when we examine the distribution entropy across boot vectors, and not just individual test measurements. Unlike most other ARM SoC we tested, the FriendlyARM occasionally produces identical boot vectors on multiple independent boots. The two most common vectors each appear 15 times in the dataset, giving a min-entropy of 11.59 bits. In other words, a sufficiently prepared adversary, given a single guess, can correctly predict the FriendlyARM's boot vector with probability $2^{-11.59}$, or about 1 in 3,000. Given 233 guesses, this probability rises to $2^{-4.7}$, or about 1 in every 25. However, this probabilistic defect does not render our instrumentation worthless. Fifty-five percent of vectors in the data set are unique, meaning that this method can fully protect the Linux randomness generator on the FriendlyARM over half the time, for a negligible cost during kernel initialization. Even if the machine does boot with a more common state, mixing in these measurements can never reduce the amount of entropy available to the pool, and thus will never be harmful to the system as a whole.

One might hypothesize that there is some "common" vector, and the other popular vectors are simply approximations thereof. However, the two most popular vectors differ in 59 of 77 positions. Also, strangely, the Mini6410 contains the same ARM core as the Raspberry Pi, which exhibits none of these symptoms. We can find no convincing explanation for the observed difference between these two systems.

8) Cubox: The Cubox is a commercially available desktop platform, powered by the Marvell ARMADA 510 SoC with an 800 MHz Sheeva ARMv7 superscalar CPU core. We modified a Linux 3.6.9 kernel for the Cubox[4], as before, to include 78 tests. We then rebooted the Cubox 27,421 times.

Per-test distribution entropy for the Cubox is presented in Table I. Interestingly, it is our only ARM SoC which has constant-time tests, i.e., tests whose distribution entropy is zero. It also presents less test entropy overall, with a sum of only 129.15 bits of individual test distribution entropy.

Like the FriendlyARM, the Cubox creates non-unique boots; the most common of these occurs 80 times (0.29%). Only 7,857 boots are unique in our data set. The total empirical distribution entropy of the data set is 12.53 bits, which indicates that our technique, while not solving the entropy-at-boot problem on the Cubox, will still help protect the kernel's entropy generation.

9) Linksys WRT54GL: While ARM-based embedded devices and SoCs are becoming more and more popular, any investigation into entropy on embedded devices would be remiss without examining how well proposed techniques apply to the large installed base of devices. Home routers, which were recently shown to have insufficient entropy for certificate generation [16], represent an enormous number of existing devices, and, perhaps more importantly, embedded devices where strong entropy sources are extremely important (e.g., key generation). To examine these routers, we chose the Linksys WRT54GL as our test platform.

[4]Online: https://github.com/rabeeh/linux.git

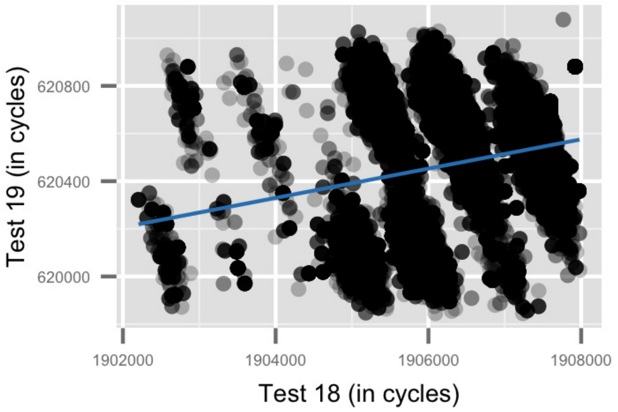

Figure 7: Cycle counts for Tests 18 and 19 on a WRT54GL. Each point is one boot. Line is best-fit linear model.

The WRT54GL is a popular consumer 802.11B/G wireless router, and consists of a Broadcom BCM5352 "router-on-a-chip", which contains a 200 MHz MIPS32 core; 16 MiB of RAM; and 4 MiB of flash. Importantly for our purposes, Linksys provides a custom Linux 2.4.20 kernel which can be modified and run on the device.

The stripped-down WRT54GL kernel has fewer function calls in `start_kernel` than the more modern kernels on our ARM boards, but this is to be expected given the simplicity of the device: the kernel needn't contain any extraneous code. We are able, then, to insert 24 tests in the kernel initialization.

We then ran our modified kernel on two separate WRT54GLs, one unmodified at 200 MHz and one overclocked to 250 MHz. The unmodified WRT we rebooted 81,057 times, while we rebooted the overclocked device 54,465 times. The per-test distribution entropies for the unmodified device are in Table I. Perhaps surprisingly for this device, these per-test entropies are quite high, up to 10 bits in some cases.

However, the correlations between tests on the WRT54GL are far more intertwined than they are on our preceding ARM devices. Two plots of these correlations can be seen in Figures 7 and 8.

Unfortunately, the overall entropy performance of the WRT54GL betrays its promising per-test entropies. Across the 81,057 boots of our unmodified router, we only see 11.86 bits of distribution entropy, and the most common boot sequence appears 1,247 times (10.4%). Indeed, the top 188 vectors make up 37.1% of the dataset (30,062 boots). If this were the only source of entropy for a PRNG seed, a motivated attacker could easily brute-force these few vectors and succeed almost 40% of the time. Even worse, there are

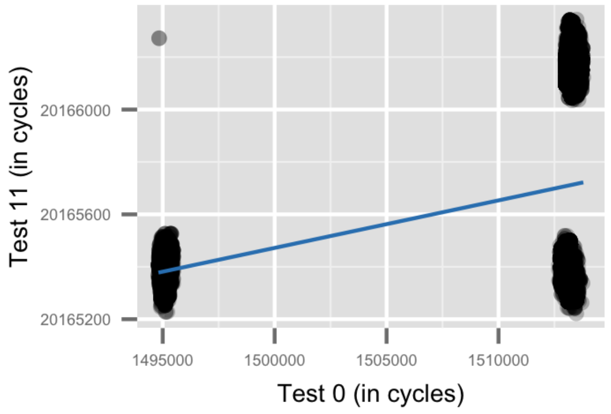

Figure 8: Cycle counts for Test 0 and 11 on an WRT54GL. Each point is one boot. Line is best-fit linear model.

only 11,960 boot sequences we saw only once. If the attacker simply checks the 4,209 vectors that she saw more than once during her precomputation, she will succeed against 78.6% of boots.

This unfortunate distribution shows that boot–time entropy is insufficient to protect a PRNG on a standard MIPS home router. However, it does add somewhat more than 11.86 bits, which is our observed distribution entropy across the 24-element test result vectors. Since the process relies solely on an already–extant hardware counter and is virtually free, adding it to the Linux kernel boot is still worthwhile.

Overclocking: To see if we could tease further entropy from the WRT54GL, we tried overclocking it from 200 MHz to 250 MHz, on the theory that we could change the ratios between clocks in different parts of the SoC and RAM. On this modified device, we performed 54,465 reboots. Overclocking does materially change each test's distribution: the Kolmogorov-Smirnov test for distribution equality reports $D > 0.1, P < 2.2 \cdot 10^{-16}$ for 19 of 24 tests, indicating that the two device's empirical test values are drawn from different underlying distributions. However, the overclocked processor shows the same type of grouping as the unmodified system, giving only 10.4 bits of distribution entropy over the 24-element boot vectors, with the most common appearing 879 times (1.6%).

10) Atmel NGW100 mkII: Finally, we turn to the Atmel NGW100 mkII, a development board for the AT32AP7000-U AVR32 microprocessor. AVR32 processors are designed to be small, low-cost, and low-power: in some sense, it's one of the smallest microprocessors capable of running Linux. Designed to prototype network gateway devices, the NGW100 mk II ships with multiple Ethernet connectors, internal flash storage, and an SD card slot. To maintain consistency, we booted the NGW100 mkII off an SD card.

We modified and built a patched Linux 2.6.35.4 kernel using the Atmel AVR32 Buildroot system, adding 69 tests to `start_kernel`. Then, via an RS-232 console, we rebooted the board 38,157 times. The per-test distribution entropy can be found, as usual, in Table I.

As befits our hypothesis that simpler processors produce more constant results, 28 of the 69 tests have absolutely no variation at all. Most of these functions are simply empty, as the AVR32 is simple enough to not need their services (e.g., `setup_nr_cpu_ids` is a no-op, as there are no multi-core AVR32 systems), but others do various memory initialization tasks. The constant execution time of these functions speaks to the minimal nature of the system as a whole.

Perhaps not surprisingly, this simplicity takes a toll on the entropy generated during the boot process. Indeed, in our data set, we see only 418 unique 69-element boot vectors; the least frequent of which appears 43 times (0.1%), while the most frequent appears 314 times (0.8%). This suggests rather strongly that we have collected every possible variation of test times the device will generate under standard operating conditions. The empirical distribution entropy of our data set is 8.58 bits; this is likely all the entropy that can be extracted from timing the NGW100 mkII boot.

III. Architectural Causes of Timing Variation

In this section, we describe two physical mechanisms that partly explain the non-determinism we measured during the execution of early kernel code: communication latency (variation that can arise while sending data between two clock domains) and memory latency (variation that arises due to interactions with DRAM refresh). We give evidence that these mechanisms are involved. We stress that these two mechanisms only partly explain the behavior we observed. Other mechanisms we do not understand are likely also involved; we hope that future work can shed more light on the situation.

A. *Clock domain crossing*

Modern embedded processors contain multiple clock domains, and due to misalignment between the domains, the amount of time it takes to send messages between two clock domains can vary.

Processor designs use multiple clock domains to allow different portions of the chip to run at different frequency. For instance, on the BeagleBoard xM, the ARM Cortex-A8 runs at 1 GHz, the peripheral interconnect runs at 200 MHz, and the Mobile DDR memory runs at 166 MHz [36].

At each boundary between two domains, chip designers must use specialized mechanisms to ensure reliable communication. The most common solution to this problem is an asynchronous FIFO (or queue) that enqueues data according to one clock and dequeues it according to a second clock.

To see how an asynchronous FIFO can give rise to latency variation, consider the case when the FIFO is empty and the output domain tries to dequeue data at the same moment the input domain inserts data. If the input domain's clock

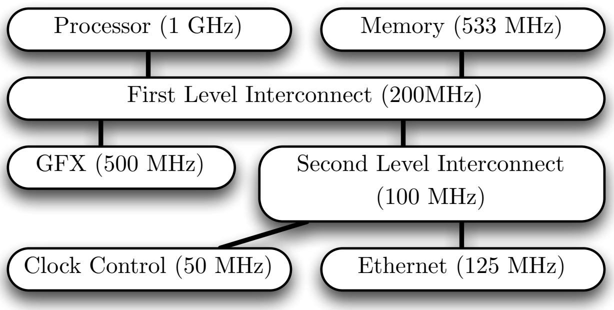

Figure 9: Clock domains similar to the domains found on the DM3730. In order for the processor to modify a register in the Ethernet controller, it must cross the clock domains of the first and second level interconnects.

arrives first, the dequeue will succeed. If the output domain's clock arrives first, the dequeue fails and will occur one clock period later. If they arrive at precisely the same time, metastability can result, which will also result in delay. Because of random (and mostly independent) variation in when the two clock signals arrive at the asynchronous FIFO (i.e., clock jitter), any of these orderings is possible and communication latency will vary.

Interactions between different clocks and metastability are well-known sources of very high-quality randomness [5, 25, 34], so it is tempting to try to exploit the domain crossing that already exist in a design to generate random bits.

In order to observe to interactions between clocks on our device, we instrumented code to measure the latency of communication between different clock domains. On the BeagleBoard xM, there are two on-chip-buses that connect peripherals, similar to the diagram on Figure 9. The processor, peripherals and interconnects are clocked by several different PLLs. For the processor to communicate with peripherals on the SoC, the processor must cross these clock domains. Our approach was to measure the variation in latency in communication devices with an increasing number of clock domain crossings. Specifically, we measured the number of cycles it took to perform to complete a set of instructions which did not cross the interconnect (two NOP instructions), to cross the first level interconnect (reading the revision number register of the memory controller) and to cross the second level interconnect (reading the revision number register of the system clock controller).

Our results are shown in Figure 10. Variability in frequency increases with the number of clock domains crossed. At two clock domain crossings, the distribution is bimodal. While there may be some serial correlation between repeated runs, this indicates that a read from the second level interconnect can provide up to around 2 bits of entropy. Reads from this register are also fast: at an average of 270 cycles, millions of these reads can be performed each second.

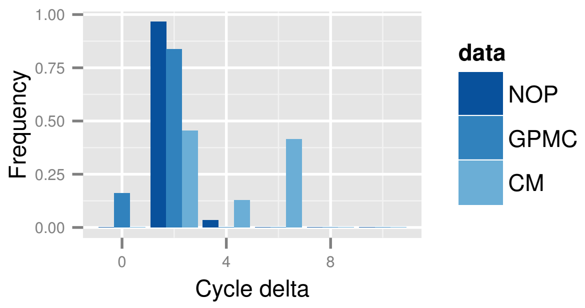

Figure 10: Execution latency for two NOP instructions (NOP), a read from the general purpose memory controller (GPMC), and a read from the clock manager (CM). Cycle delta is the difference from the minimum cycles observed.

B. DRAM Access Latency

A second source of variation in performance is interactions between main memory (i.e., DRAM) accesses, DRAM refresh, and the memory controller. Because DRAM bits decay over time, the system must periodically read and re-write each DRAM storage location. Depending on the system, the processor's memory controller issues refresh commands or, alternately, the processor can place the chips in an auto-refresh mode so they handle refresh autonomously.

Regardless of who manages it, the refresh process cycles through the DRAM row-by-row, and an incoming memory access (e.g, to fetch part of the kernel for execution) may have to wait for the refresh to complete.

To measure the effect of refresh on execution timing, we used hardware performance counters to measure the number of cycles it took to execute a series of 64 NOP instructions on a ARM Cortex-A9 [39] with the instruction cache disabled 100,000 times. We then used a software register to turn refresh off and performed the test again. The results of our test are plotted in Figure 12. The variation in execution latency was much greater with refresh turned on: with refresh on, execution fell into 6 bins, with ≈80% distributed at the mode and ≈20% distributed in the other 5 bins. With refresh off, over 99% of executions fell into the mode with less than 1% distributed in two other bins.

While refresh itself may appear to induce random distributions in our experiment, the state machines in the memory controller and the DRAM chips that manage refresh are deterministic, as is the execution of code on the CPU that generates requests. If the rest of the system were deterministic as well, we expect that DRAM accesses would have deterministic latencies.

However, other sources of randomness can affect the relationship between the processor and the DRAM refresh state machines. For instance, the PLL for the DRAM controller may "lock" more quickly than the processor's PLL at system (see $\textcircled{4}$ in Figure 11) boot or the DRAM controller's power supply may take longer to stabilize at start up (see $\textcircled{1}$ in

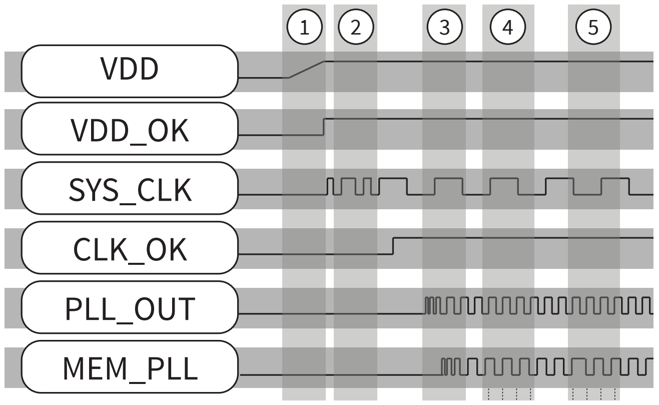

Figure 11: Power and clocks on the startup of a typical embedded system. At (1), the voltage is ramped up until it is stable, which can take a variable amount of time. At (2), The system oscillator is turned on and takes a variable amount of time to stabilize. At (3), the PLLs that source the high frequency clocks for the processor (PLL_OUT) and memory (MEM_PLL) are turned on and take a variable amount of time to stabilize. At (4), the time that the memory clock and processor clocks cross is variable but fully determined by the time that both PLLs stabilize. At (5), a small amount of jitter in the memory clock causes the position the clocks cross to change.

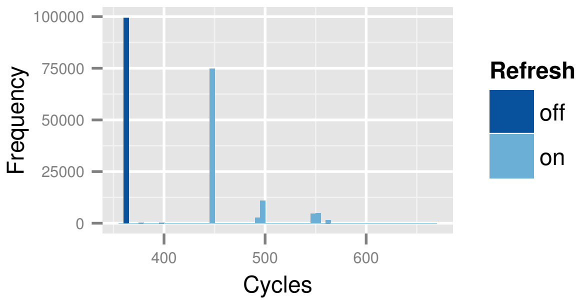

Figure 12: Execution latency with refresh on and off.

Figure 11). In this case, the future interactions between the processor and refresh state machine will be affected, and the latency for DRAM accesses will vary slightly. In addition to variation in the system's initial conditions, randomness from clock domain crossing can further perturb the interaction between the processor and memory.

IV. DRAM Decay

Ultimately, the most useful source of randomness we found in these system is the decay of data stored in DRAM over time. DRAM decay occurs when the charge that stores a binary value leaks off the capacitor in a DRAM storage cell. This phenomenon is well-studied, and the refresh process (described in III-B) is designed to mitigate it.

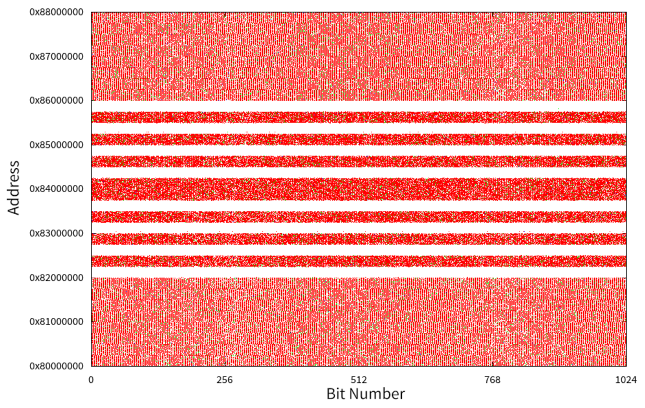

Figure 13: Decay of DRAM after 7 (Blue), 14 (Green), 28 (Yellow) and 56 (Red) seconds.

A. Disabling Refresh

In order to detect decay in a live system, we must prevent the system from refreshing DRAM. The ability to disable refresh on the memory controller is not an exotic feature: Nearly every memory controller we looked at supported disabling refresh, and every embedded SoC we looked at, from the Broadcom BCM5352 found in the WRT54GL to the DM3730 on the BeagleBoard xM had software tunable parameters for controlling refresh [36, 39]. Typically, control over refresh is used to implement sleep modes. When a processor enters a sleep mode, it disables refresh on the memory controller and sends a command to the DRAM chips to enter "self-refresh" mode, forcing the DRAM chips refresh themselves as needed. By turning off refresh on the memory controller and not sending the "self-refresh" command, we were able to observe decay in our test systems.

B. Decay

The decay rate of DRAM bits varies widely (a fact exploited by "cold boot" techniques [13]) as a result of manufacturing variation, temperature, the data stored in the cell, and other factors. In our experiments, some bits will decay quickly (e.g., on the order of hundreds of µs) while others will retain their data for seconds or hours. We find that the rate at which bits decay varies with even small variations in temperature (see Section IV-D2).

C. Experimental Setup

Our approach to harvesting randomness from DRAM is as follows. Very early in the boot process (i.e., in the boot loader) we write test data to a portion of DRAM, and then disable the refresh mechanism in both the processor's memory controller and the DRAM chips. The processor then waits several seconds, reads the data back, and XORs it with pattern written initially. Any flipped bits will appear at this stage. After this process, the bootloader can re-enable refresh, reinitialize DRAM, and continue loading as normal.

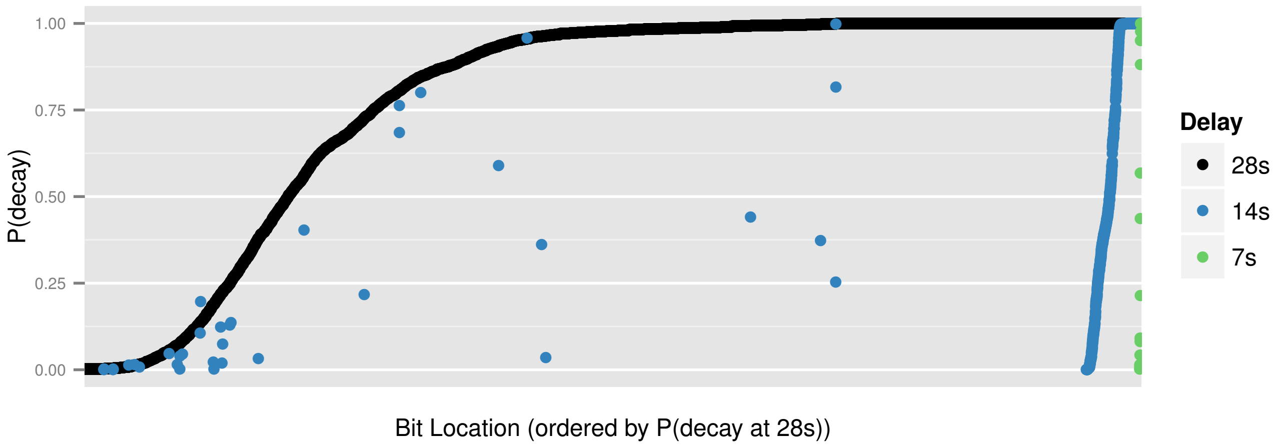

Figure 14: Probability of decay, per bit, for non-refresh times of 7s, 14s, and 28s. Ordered by Pr(decay) at 28s.

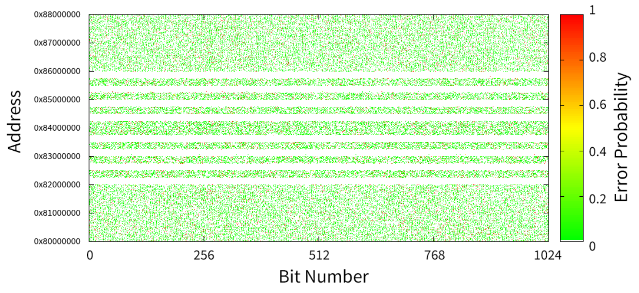

Figure 15: Probability of decay after one minute.

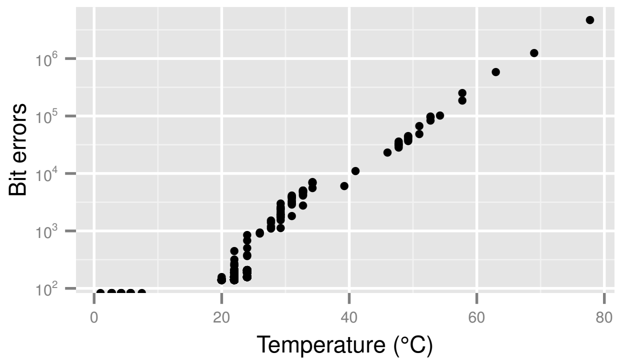

Figure 16: Relationship between temperature and DRAM decay over a constant period.

Next, we modified both layers of U-Boot, as well as the Linux kernel, to incorporate the generated entropy into the kernel randomness pools. We use a custom extracting hash to condense the memory decay into 1,024 bits, and pass the result into the kernel as a base-64-encoded parameter. Overall, hashing and processing takes less than second, on top of the unavoidable multiple-second DRAM decay time.

D. Results

1) Decay Probability and Distribution: Although we could reliably observe many bits decaying, the distribution of decay was not uniform. Figure 15 shows the distribution of decay probabilities at 58 seconds. The values range from 0 (white) to very low (green) to near certainty (red). The figure also shows that some areas of the DRAM do not appear to decay at all.

The horizontal bands in the figure are due to the test pattern initially written to memory. We wrote 0xAA to the top quarter of memory, 0x00 to the next quarter, 0xFF to the next, and 0x55 to the last quarter. In the areas which show no decay, the pattern (0x00 or 0xFF) matched the cell's "ground state" (i.e., the state into which the cell naturally decays). This can vary because chips use different voltages for "0" and "1" in different portions of the chip.

Figure 13 shows decay over time. The yellow bits decayed first and the red bits decayed last. Unsurprisingly, the longer the interval, the more bit errors occur and the more randomness we are able to extract. In Figure 14, each bit's probability of decay over 7, 14, and 28 seconds has been graphed. Perhaps unsurprisingly, every bit that ever decays within 7 seconds has a 100% chance of decaying in 14 or 28 seconds. Interestingly, a number of bits with a non-zero probability of decaying in 14 seconds don't always decay by 28 seconds, indicating that DRAM bits don't simply decay in a set order, and can provide true entropy.

2) Temperature Dependence: Previous work has shown that decay varies with DRAM temperature. To compensate, many systems increase the refresh frequency as temperature increases [21, 35]. Primarily, this is due to the increase in DRAM cell leakage as temperature rises [21]. To understand the effect of this temperature dependence on the probability of decay, we set up an experimental protocol that allowed us to control the temperature of DRAM. By submerging

DRAM in non-conductive isopropyl alcohol inside a refrigerator and using an aquarium heater, we were able to control the DRAM temperature to $\pm 1^\circ$ C. For temperatures above 35° C, we used a laboratory oven to measure decay at high temperatures.

Our results are shown in figure 16. We find that at low temperatures, the few bits which decay are consistent (i.e. the same bits always decay). Around 20° C, we begin to see bits that sometimes decay. At room temperature (23° C), we begin to see an exponential rise in bit decay.

3) DRAM Temperature Modification: We can generate more randomness by increasing the temperature of the DRAM chip. To accomplish this we wrote a simple ‘power virus’ that attempts to maximize DRAM power consumption and heat dissipation. The virus initializes a region of DRAM and then reads repeatedly from different addresses. The initial data and the addresses are choosen to maximize the transition frequency on the DRAM pins.

We find that by implementing our power virus, we heat up the DRAM from 26° C to 29° C within 1 minute. We run the power virus while waiting for bits to decay.

4) Variability: In addition to randomness in bit decay between boots, we also observed two kinds of variability between individual boards: Decay probability variability, the variability in the probability that different bits will decay; and cold state variability, the variability in the initial contents of DRAM after a cold boot.

This variability is due to manufacturing variations that cause DRAM cells to leak at different rates, leading to the decay probability variability we observe. Process variations in the sense amplifiers (which convert the analog values from the DRAM bits into logical “0”s and “1”s) is also well documented [15], and probably contributes as well.

The variation in the DRAM’s contents from a cold boot (measured after the device was powered off for 3 days) can provide a unique fingerprint for each board. For instance, at $25-28^\circ$ C with a delay of 7s, on one BeagleBoard, a certain 10 bits always decay, while the other BeagleBoard has only 6 such bits. The two sets are disjoint; that is, the bits that decay on one board do not decay on the other.

Under the assumption that, due to process variation, the placement of these “leaky” bits is independent between different DRAM modules, the locations of leaky bits act as a fingerprint for a particular BeagleBoard. Verifying this assumption about the distribution of leaky bits would require access to more systems than we have, and we leave it to future work.

E. Extracting per-boot randomness from DRAM

The location of leaky bits cannot, by itself, be the basis for random number generation. An attacker who has physical access, who can mount a remote code-injection exploit, or can otherwise run software on the device will be able to locate its leaky bits. Therefore, we must examine the bits that sometimes decay and sometimes do not.

We now give a rough estimate for the amount of entropy available in the decay of these bits. Our analysis makes the assumption that bits decay independently of each other. This is a strong assumption and there evidence that it is at least partly false, e.g., Section 3.3 of [13]. Accordingly, the entropy estimates below are overestimates. We hope future work can provide a better measure of available entropy.

We estimate Pr[decay] for each bit based on our experiments and use this probability to compute the information theoretic entropy content for this bit:

$$E(p) = -\big(p \cdot \lg(p) + (1-p) \cdot \lg(1-p)\big) \qquad (3)$$

Under the assumption that bits decay independently of each other, we can simply sum this distribution entropy over every bit we saw decay.

For a BeagleBoard xM at 25-27°C and with a decay time of 7 s, we obtain a total boot-time entropy estimate of 4.9 bits, largely due to the fact that only 19 memory decays ever happen, and 16 of these happen with $p > 0.9$ or $p < 0.1$. For a decay time of 14s, we see 511 bits ever decay, and summing their entropy contributions gives an entropy estimate of 209.1 bits. For a delay of 28s, 9,943 bits decay, for an estimated entropy of 8,415.16 bits. For 56 seconds, we see 427,062 decays, for an estimated entropy of 98,611.85 bits.

A delay of even 14s on first boot is unacceptable in many applications. Moreover, because DRAM decay depends on temperature, this approach may not provide security in very cold conditions — for example, phones used on a ski slope.

V. PLL Lock Latency

The PLLs that produce the on-chip clocks in modern processors are complex, analog devices. When they start up (or the chip reconfigures them), they take a variable amount of time to “lock” on to the new output frequency (see $\textcircled{3}$ in Figure 11). This variation in lock time is due to a number of factors, including stability of the power supply, accuracy and jitter in the source oscillator, temperature, and manufacturing process variation [17]. Repeatedly reconfiguring an on-chip PLL and measuring how long it takes to lock will result in random variations.

SoCs typically contain several PLLs used to derive clocks for the processor, memory and peripherals. On the BeagleBoard xM, the DM3730 contains 5 DPLLs (Digital Phase Locked Loops). Each DPLL can be reconfigured and toggled via a software register, and a status flag and interrupt will signal when a DPLL is locked. To measure the time it takes to acquire a lock, we instrumented code to disable the DPLL for the USB peripheral clock on the BeagleBoard xM. Using the hardware performance counter, we measured the number of cycles it took for the DPLL to reacquire a lock (Figure 17).

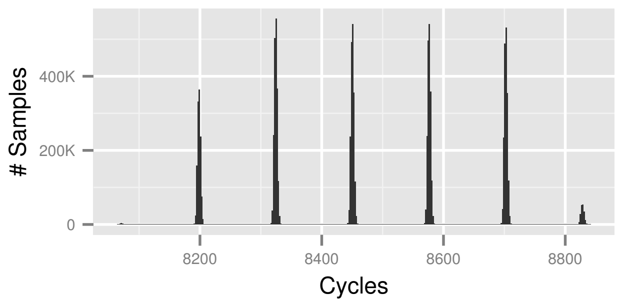

Figure 17: DPLL lock latency histogram measured by the performance counter on the BeagleBoard xM.

We obtain about 4.7 bits of entropy every time we re-lock the DPLL, and it takes at most approximately 9000 cycles (9µs) for the DPLL to re-lock. Using the DPLL lock latency, we can obtain about 522 KiB of pure entropy per second.

DPLL lock latency could be easily polled for entropy during early boot when the SoC first sets up the clocks and PLLs in the system. Since the DPLL is affected by analog conditions such as temperature, a determined attacker may be able to induce bias in the lock time.

VI. Conclusion

Randomness is a fundamental system service. A system cannot be said to have successfully booted unless it is ready to provide high-entropy randomness to applications.

We have presented three techniques for gathering entropy early in the boot process. These techniques provide different tradeoffs along three metrics: how high the bitrate, how specific to a particular system, and how well explained by unpredictable physical processes.

Our first technique, which times the execution of kernel code blocks, provides a moderate amount of entropy and is easily applied to every system we examined, but we are able to give only a partial account for the source of the entropy it gathers.

Our second technique, DRAM decay, provides a large amount of entropy, but presents a heavy performance penalty and is tricky to deploy, relying on details of the memory controller. Its advantage is a physical justification for the observed randomness.

Our third technique, timing PLL locking, promises the highest bitrate and is well supported by physical processes, but its implementation requires intimate knowledge of the individual SoC.

We implemented and characterized these techniques on a broad spectrum of embedded devices featuring a variety of popular SoCs and hardware, from resource-rich mobile phone hardware to devices that aren't much more than an ethernet port and a SoC. While these three techniques certainly can be applied to traditional desktop systems as well as more powerful embedded devices, in some sense, our tiny embedded systems start at a disadvantage. Wireless devices can read randomness from radios; desktops can rely on saved entropy from previous boots. Our work focuses on adequately protecting headless, resource-poor embedded devices, which must acquire strong entropy on their very first boot, before they can even export network connectivity.

Our work leaves many questions open. We are able to give only a partial explanation for the entropy we observed in our first technique, and only a partial characterization of the DRAM decay effects in our second technique. We hope that future work can shed more light on the situation. More work is also necessary to understand how much the gathered entropy depends on environmental factors that might be under adversarial control.

The three techniques we present exploit just a few of the many potential architectural sources of randomness available in modern systems. Other possible sources of entropy, which we hope will be explored in future work, include voltage scaling latency, GPIO pin voltage, flash memory corruption patterns, and power supply stabilization latency.

Our three techniques are all, ultimately, workarounds for the lack of dedicated hardware random number generators in embedded architectures. What will spur the adoption of such hardware, by both hardware and software developers? What is the right way to specify such hardware for the ARM architecture, where a high-level core description is licensed to many processor manufacturers? Furthermore, is it possible to verify that such a unit is functioning correctly and free of backdoors?

Acknowledgments

We thank Daniel J. Bernstein, J. Alex Halderman, Nadia Heninger, and Eric Rescorla for their comments and suggestions.

We would like to thank Xilinx for donating hardware.

This material is based upon work supported by the National Science Foundation under Grants No. CNS-0831532, CNS-0964702, DGE-1144086, and by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

References

[1] E. Barker and J. Kelsey, "Recommendation for random number generation using deterministic random bit generators," NIST Special Publication 800-90A, Jan. 2012, online: http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf.

[2] M. Bellare, Z. Brakerski, M. Naor, T. Ristenpart, G. Segev, H. Shacham, and S. Yilek, "Hedged public-key encryption: How to protect against bad randomness," in *Asiacrypt 2009*. Springer, Dec. 2009.

[3] J. Bouda, J. Krhovjak, V. Matyas, and P. Svenda, "Towards true random number generation in mobile environments," in *NordSec 2009*. Springer, Oct. 2009.

[4] E. Brickell, "Recent advances and existing research questions in platform security," Invited talk at Crypto 2012, Aug. 2012.

[5] J.-L. Danger, S. Guilley, and P. Hoogvorst, "High speed true random number generator based on open loop structures

in FPGAs," *Microelectronics Journal*, vol. 40, no. 11, Nov. 2009.

[6] D. Davis, R. Ihaka, and P. Fenstermacher, "Cryptographic randomness from air turbulence in disk drives," in *Crypto 1994*. Springer, Aug. 1994.

[7] L. Dorrendorf, Z. Gutterman, and B. Pinkas, "Cryptanalysis of the random number generator of the Windows operating system," *ACM Trans. Info. & System Security*, vol. 13, no. 1, Oct. 2009.

[8] D. Eastlake 3rd, S. Crocker, and J. Schiller, "Randomness Recommendations for Security," RFC 1750 (Informational), Internet Engineering Task Force, Dec. 1994, obsoleted by RFC 4086. [Online]. Available: http://www.ietf.org/rfc/rfc1750.txt

[9] V. Fischer and M. Drutarovský, "True random number generator embedded in reconfigurable hardware," in *CHES 2002*. Springer, 2003.

[10] I. Goldberg and D. Wagner, "Randomness and the Netscape browser," *Dr. Dobb's Journal*, Jan. 1996.

[11] P. Gutmann, "Software generation of practically strong random numbers," in *USENIX Security 1998*. USENIX, Jan. 1998.

[12] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the Linux random number generator," in *IEEE Security and Privacy ("Oakland") 2006*. IEEE Computer Society, May 2006.

[13] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *USENIX Security 2008*. USENIX, Jul. 2008.

[14] M. Hamburg, P. Kocher, and M. E. Marson, "Analysis of Intel's Ivy Bridge digital random number generator," Online: http://www.cryptography.com/public/pdf/Intel_TRNG_Report_20120312.pdf, Mar. 2012.

[15] R. Heald and P. Wang, "Variability in sub-100 nm SRAM designs," in *ICCAD 2004*. IEEE Computer Society, Nov. 2004.

[16] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of widespread weak keys in network devices," in *USENIX Security 2012*. USENIX, Aug. 2012.

[17] P. Heydari, "Analysis of the PLL jitter due to power/ground and substrate noise," *IEEE Trans. Circuits and Systems I*, vol. 51, no. 12, Dec. 2004.

[18] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-up SRAM state as an identifying fingerprint and source of true random numbers," *IEEE Trans. Computers*, vol. 58, no. 9, Sep. 2009.

[19] A. Hubert and R. van Mook, "Measures for Making DNS More Resilient against Forged Answers," RFC 5452 (Proposed Standard), Internet Engineering Task Force, Jan. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5452.txt

[20] M. Jakobsson, E. Shriver, B. K. Hillyer, and A. Juels, "A practical secure physical random bit generator," in *CCS 1998*. ACM, Nov. 1998.

[21] *DDR3 SDRAM Standard JESD79-3F*, JEDEC Committee JC-42.3, Jul. 2012, online: www.jedec.org/sites/default/files/docs/JESD79-3F.pdf.

[22] D. Kaminsky, "Black ops 2008: It's the end of the cache as we know it," Black Hat 2008, Aug. 2008, presentation. Slides: https://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Kaminsky/BlackHat-Japan-08-Kaminsky-DNS08-BlackOps.pdf.

[23] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Cryptanalytic attacks on pseudorandom number generators," in *FSE 1998*. Springer, Mar. 1998.

[24] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator," in *SAC 1999*. Springer, 2000.

[25] P. Kohlbrenner and K. Gaj, "An embedded true random number generator for FPGAs," in *FPGA 2004*. ACM, Feb. 2004.

[26] P. Lacharme, A. Röck, V. Strubel, and M. Videau, "The Linux pseudorandom number generator revisited," Cryptology ePrint Archive, Report 2012/251, 2012, http://eprint.iacr.org/.

[27] N. McGuire, P. O. Okech, and Q. Zhou, "Analysis of inherent randomness of the Linux kernel," in *RTLW 2009*. OSADL, Sep. 2009, online: http://lwn.net/images/conf/rtlws11/random-hardware.pdf.

[28] T. Mytkowicz, A. Diwan, and E. Bradley, "Computer systems are dynamical systems," *Chaos*, vol. 19, no. 3, Sep. 2009.

[29] N. Nisan and A. Ta-Shma, "Extracting randomness: A survey and new constructions," *J. Computer and System Sciences*, vol. 58, no. 1, Feb. 1999.

[30] C. Pyo, S. Pae, and G. Lee, "DRAM as source of randomness," *Electronics Letters*, vol. 45, no. 1, 2009.

[31] T. Ristenpart and S. Yilek, "When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography," in *NDSS 2003*. Internet Society, Feb. 2003.

[32] 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," NIST Special Publication 800-22, Revision 1a, Apr. 2010, online: http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf.

[33] A. Seznec and N. Sendrier, "HAVEGE: A user-level software heuristic for generating empirically strong random numbers," *ACM Trans. Modeling & Computer Simulation*, vol. 13, no. 4, Oct. 2003.

[34] B. Sunar, W. J. Martin, and D. R. Stinson, "A provably secure true random number generator with built-in tolerance to active attacks," *IEEE Trans. Computers*, vol. 56, no. 1, Jan. 2007.

[35] M. Technology, *MT41J256M4 DDR3 SDRAM Datasheet, Rev. I*, Feb. 2010, online: http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf.

[36] *AM/DM37x Multimedia Device Silicon Revision 1.x Version R Technical Reference Manual*, Texas Instruments, Sep. 2012, online: http://www.ti.com/lit/ug/sprugn4r/sprugn4r.pdf.

[37] The Debian Project, "openssl – predictable random number generator," DSA-1571-1, May 2008, http://www.debian.org/security/2008/dsa-1571.

[38] J. Voris, N. Saxena, and T. Halevi, "Accelerometers and randomness: Perfect together," in *WiSec 2011*. ACM, Jun. 2011.

[39] *Zynq-7000 All Programmable SoC Technical Reference Manual, Version 1.3*, Xilinx, Oct. 2012, online: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[40] S. Yilek, "Resettable public-key encryption: How to encrypt on a virtual machine," in *CT-RSA 2010*. Springer, Mar. 2010.

[41] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, "When private keys are public: Results from the 2008 Debian OpenSSL vulnerability," in *IMC 2009*. ACM, Nov. 2009.