ERHARD-RNG: A Random Number Generator Built from Repurposed Hardware in Embedded Systems

Jacob Grycel and Robert J. Walls
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA
Email: jtgrycel, rjwalls@wpi.edu

Abstract—Quality randomness is fundamental to cryptographic operations but on embedded systems good sources are (seemingly) hard to find. Rather than use expensive custom hardware, our ERHARD-RNG Pseudo-Random Number Generator (PRNG) utilizes entropy sources that are already common in a range of low-cost embedded platforms. We empirically evaluate the entropy provided by three sources—SRAM startup state, oscillator jitter, and device temperature—and integrate those sources into a full Pseudo-Random Number Generator implementation based on Fortuna [1]. Our system addresses a number of fundamental challenges affecting random number generation on embedded systems. For instance, we propose SRAM startup state as a means to efficiently generate the initial seed—even for systems that do not have writable storage. Further, the system’s use of oscillator jitter allows for the continuous collection of entropy-generating events—even for systems that do not have the user-generated events that are commonly used in general-purpose systems for entropy, e.g., key presses or network events.

I. INTRODUCTION

Modern security technologies depend on strong random numbers for creating encryption keys, signatures, and nonces for sensitive data. The strength of these numbers is dependent on a reliable source of entropy and a secure Pseudo-Random Number Generator (PRNG). On general purpose systems such as Linux, the entropy is usually gathered from the timing of unpredictable events, e.g., user key strokes, mouse movements, and disk events. Further, initial seeding of the generator is usually done by reading in a seed file or by delaying random number generation until enough entropy can be collected. However, entropy sources used by general purpose systems are poorly suited to the embedded environment. For example, key stroke information cannot be collected by a system that does not have a keyboard attached. Similarly, seed files cannot be used when the system does not have a disk.

This work examines the feasibility of implementing a PRNG fed by high-quality entropy sources in an embedded environment with limited hardware and software resources. Previous studies proposed using specialized hardware in either an FPGA or an ASIC to gather entropy [2, 3]; However, such hardware can add significant production cost to embedded systems. In contrast, the system we present, ERHARD-RNG, utilizes existing internal hardware that is commonly found in a wide variety of inexpensive microcontrollers. In particular, our implementation of ERHARD-RNG targets the Texas Instruments MSP430 (MSP430F5529) low-power microcontroller using only an external 4 MHz crystal oscillator and power supply. We chose this platform due to its numerous programmable peripherals, its commonality in embedded systems development, and its affordability. Many microcontrollers and Systems On a Chip (SoC) include similar hardware that allows our PRNG to be implemented, including chips from Atmel, Xilinx, Cypress Semiconductor, and other chips from Texas Instruments.

ERHARD-RNG is based on three key insights. First, we can address the challenge of collecting entropy at runtime by sampling the jitter of the low-power/low-precision oscillator. Coupled with measurements of the internal temperature sensor we can effectively re-seed our PRNG with minimal overhead. Second, we can solve the problem of initial seeding by leveraging randomness inherent in the startup state of RAM [4]. Finally, we also propose the use of a Cyclic Redundancy Check (CRC) as a mixing and collapsing function for transforming the RAM startup state into the initial seed, overcoming the processing limitations created by a small memory space.

Our system, based on the Fortuna PRNG [1], was designed and implemented on the MSP430. We empirically evaluate the amount of entropy generated by each of our sources and the overall quality of the random numbers generated by the PRNG. Additionally, we analyze the effect of different operating environments on the entropy sources; As systems operate at different supply voltages and environment temperatures, it is important to understand the extent to which the performance of the PRNG is dependent on particular physical parameters, e.g., external temperature.

II. DESIGN AND IMPLEMENTATION OF ERHARD-RNG

PRNGs rely on a source of true randomness, entropy, that often provides an initial seed from which a random stream of data is produced. Depending on the generator design, entropy sources may also be used to periodically re-seed the PRNG, extending the number of outputs that can be produced before the generator is liable to repeat sequences or become predictable. One such generator that frequently uses entropy sources to re-seed is Fortuna, a block cipher-based PRNG designed by Niels Ferguson and Bruce Schneier [1]. Fortuna forms the foundation of ERHARD-RNG.

ERHARD-RNG, like Fortuna, is divided into two components: the generator and the accumulator. The generator consists of a block cipher that encrypts a counter value to...
produce a block of random bytes. After every data request the encryption key is replaced with a new output from the block cipher in order to reduce security vulnerabilities in the event of attacker learning of the internal state [11].

Beyond basic implementation details, ERHARD-RNG includes a number of design changes from the original specification of Fortuna: seed file management, entropy pool inputs, and entropy pool size.

**Seed File Management.** The originally specified manner for managing the seed file is to rewrite the contents of the seed with an output from the generator after every startup [11]. On an embedded device this is difficult due to the lack of an initial seed on the first boot of a device and limited non-volatile memory. By using SRAM startup state (see Section III-C), we removed the need for managing a seed file. Instead, ERHARD-RNG reads the mixed SRAM state from the low 64 bytes of memory, which is produced during a CRC startup routine before the main program begins.

This seed management scheme also improves the boot speed of the application when compared to the alternative of writing and reading a seed file from FLASH. FLASH operations take a considerable amount of time and would slow down the device start up during seed file management. Using the CRC mixing function is fast, as each calculation happens in hardware within 2 clock cycles while the CPU prepares the next byte of SRAM for computation.

**Entropy Pool Contribution.** The accumulator consists of a set of 32 entropy pools which are fed random data from any number of entropy sources. In Fortuna, the data contributed to each pool consists of the random data itself and a source identifier which indicates the origins of the sample. However, as memory is not readily available in embedded devices, ERHARD-RNG does not include the source IDs when adding the source sample to an entropy pool.

**Entropy Pool Size.** When a pool is filled with enough entropy to replenish the generator, the pool contents are collapsed into a 256-bit value that becomes the new generator key. The collapsing is done using a cryptographic hashing algorithm. For ERHARD-RNG, 58 events are required to reseed the generator with 128 bits of entropy; this number was used as the minimum entropy pool size for re-seeding in order to ensure consistent protection of the generator state. ERHARD-RNG’s entropy sources can re-seed the generator every 3 ms.

### A. Algorithm Selection

Fortuna’s design does not specify a particular block cipher or hashing algorithm, however there are limitations on which algorithms can be used due to key, block, and digest sizes. The block cipher is required to have a 128-bit block to fit a 128-bit counter value, and a 256-bit key. The hashing algorithm digest must also be 256-bit in order to produce new keys during re-seeding. Further limitations placed our ERHARD-RNG variation were code size and execution time due to limited memory and a slow clock speed.

### III. Entropy Sources

ERHARD-RNG leverages three components of the MSP430 as entropy sources for seeding the generator and feeding the accumulator. Two of the sources, phase jitter in an internal low-power oscillator and a temperature sensor were used as feeders to the entropy pool. The third source, the startup state of SRAM, was used as an initial seed for the generator. The following sections discuss each of these sources and analyze their behavior and entropic qualities.
A. Low-Powered Oscillator

The MSP430 has an internal Very Low-Power Oscillator (VLO) that is intended to be used in low-power applications where an external oscillator is not able to be powered or is not present. As it is powered internally and is not sourced from a high-precision crystal, the VLO is subject to phase jitter. Phase jitter is the small time difference between when a controlled oscillator has a rising edge, and when it is expected to have a rising edge in an ideal model [6]. This signal artifact has been used in other RNGs before [2], [3], [7].

One other characteristic of interest is oscillator wander, the tendency for the frequency to stray far away from the typical frequency. Unlike phase jitter, this phenomenon is not desirable in an entropy source, as oscillator wander can be influenced by changes in the operating environment [6]. Using an oscilloscope capture in infinite-persist mode we observed that the period sits at a consistent typical frequency with slight fluctuations in the form of phase jitter, i.e., the VLO instability manifests as jitter and not oscillator wander.

Entropy Estimate. We used the NIST SP 800-90B entropy estimation suite to measure a min-entropy of 1.47 bits per 8-bit sample at a voltage of 2.4 V using one million samples. We also tested samples from the VLO across a voltage range of 2.4 V to 3.6 V—the maximum supply range for the MSP430 running at 24 MHz [8]—and a temperature range of 75 °F to 0 °F. The lowest min-entropy we observed was 1.19 bits per sample at 2.9 V at 75 °F.

At a fixed temperature and voltage, the VLO min-entropy estimates remained consistent, regardless of the temperature itself. However, when samples were collected while the environment temperature was actively changing, the entropy estimates decreased. This is likely because of the VLO periods increasing or decreasing in a constant direction, introducing a more predictable pattern to the period samples.

B. Temperature Sensor

The second runtime entropy source used by ERHARD-RNG is an internal temperature sensor that measures an aggregate of the environment and internal temperatures of the microcontroller. The sensor was routed as an input to a 12-bit Analog to Digital Converter (ADC) with a reference voltage of 1.5 V to increase the resolution of the sensor to 0.360 mV/bit for maximum sensitivity. In order to allow each entropy source to contribute to the pools evenly, we configured the ADC to make conversions of the temperature sensor at 9.5 kHz to match the typical period of the VLO at a supply of 3.3 V.

C. SRAM Startup State

In addition to the runtime entropy sources, ERHARD-RNG also requires a 64-byte seed file that initializes the generator key at start up; this is a challenge in embedded devices due to limited non-volatile memory. However, previous work has shown that the startup state of SRAM cells exhibits a random pattern and can be used as a large pool of boot-time entropy in embedded systems [6]. Our ERHARD-RNG implementation utilized the start up state of SRAM on the MSP430 (10 KB) to create a 64-byte seed file. Here we define startup state as the initial memory state after power-on before any function calls or data were placed on the software stack.

Entropy Estimate. We measured the startup state across 100 startups with 30 second power-off periods between each startup to create a set of one million samples. Although SRAM was estimated to have 0.457844 bits of entropy per byte, we observed that bytes change at different rates: The majority of the bytes changed from 3 to 10 times, while others changed up to 30 times. This observation suggest that the randomness is likely spread out over the entire memory and it is important to use the entire memory to calculate the seed—i.e., 10 KB needs to be collapsed down to a 64 byte memory segment without disturbing the RAM state. To do this, ERHARD-RNG uses a built-in CRC module to mix each block of 160 bytes down to a 16-bit value with high entropy. The viability of this CRC-CCITT-16 hardware mixing function is discussed in the following section.

D. Comparison of Entropy Sources

As a performance and security benchmark, we compared the entropy sources in ERHARD-RNG to those used by a general purpose Linux system. The Linux generator uses three sources of entropy: User input events (keyboard and mouse), hardware interrupts, and disk access events [9]. Each event is paired with a timestamp, the precision of which is dependent on the system configuration; for our experiments we observed timestamps with 4 ms precision. In addition to the entropy sources, the generator’s entropy pool is saved and restored across system power cycles to provide starting entropy.

Using a modification to the Linux kernel, we collected one million samples from each source to test with the NIST estimation suite. Shown in Table II the maximum amount of entropy from a single source was 0.742 bits per sample from

| TABLE I: Randomness tests on ERHARD-RNG samples |
|-----------------------------------------------|
| Frequency         | C1  | C2  | C3  | C4  | C5  | C6  | C7  | C8  | C9  | C10 | P-Value | Pass Ratio |
|-------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---------|------------|
| BlockFrequency    | 276 | 287 | 264 | 312 | 261 | 289 | 275 | 277 | 285 | 274 | 0.660097 | 2774/2800  |
| CumulativeSums    | 292 | 242 | 293 | 301 | 274 | 255 | 267 | 253 | 320 | 303 | 0.012556 | 2769/2800  |
| Runs              | 260 | 285 | 323 | 273 | 276 | 267 | 261 | 287 | 269 | 299 | 0.205375 | 2774/2800  |
| LongestRun        | 275 | 313 | 286 | 296 | 281 | 274 | 248 | 266 | 269 | 292 | 0.314759 | 2773/2800  |
user input events. The time needed to collect the high-entropy input events spanned three days, whereas interrupts required 1 hour, and disk events required over a day for disk access events to be initiated by the user.

The ERHARD-RNG sources have significantly higher estimates than the Linux entropy sources, particularly the VLO and temperature sensor. Additionally, both the user input and disk event sources are dependent on user input, which could starve the system if the computer is not used frequently enough. In contrast, the sources used in this work are not dependent on an external user. Furthermore, the ERHARD-RNG sources run much faster than the Linux sources, and can generate a high volume of samples in a very short period of time (on the order of seconds).

1) CRC as a Mixing Function: CRCs are a type of algebraic cyclic code that operates on the underlying principles of polynomial division over the field $\text{GF}(2)$, modulo a generator polynomial $G(x)$. The individual bits of the CRC inputs are treated as the binary coefficients of polynomials, and the remainder, often called the syndrome, of division by $G(x)$ is used as the CRC output [10]. This structure of a cyclic code is important in the analysis of CRCs for entropy mixing.

One class of entropy mixing functions is resilient functions. A $[n, m, d]$ resilient function maps from a domain of $Z_2^n$ to $Z_2^m$ (n-bit and m-bit binary strings), where the knowledge or control over $d$ bits of the input reveal no information about the output. Such functions can be formed from linear error-correcting codes with Hamming Distance $d + 1$. The mapping from domain to codomain is performed by multiplying an input by a generator matrix, $G$, for the linear code $[F(x) = xG^T]$. This operation is analogous to calculation of a syndrome within a cyclic code [7], and makes the CRC-CCITT-16 an attractive candidate for a resilient function.

Since CRCs are error-correcting codes, they are designed to produce different syndromes for different input sequences to allow for error detection. As a 16 bit CRC, this variation of the code is able to detect any bit errors (ie. changing bits), within a 16-bit input [10]. Thus, there is a unique output for any inputs up to 16 bits, so individual oscillator period sample inputs will have unique effects on the final CRC result. Furthermore, as CRC-CCITT-16 is implemented as a Linear Feedback Shift Register (LFSR), the nature of the output is randomized, as is the purpose of a LFSR-based RNG [11].

Beyond the results of individual inputs, longer input streams must also produce different results even with few varying input bits. A generator with a primitive polynomial factor of degree $n$ for a CRC guarantees detection of 2-bit differences in input streams that are at most $2^n - 1$ bits apart [10]. Thus it can be guaranteed that different CRC-CCITT-16 outputs will result for input streams that differ in 2 bits that are 32767 places away from each other, far larger than the maximum of 1280 bits that pass through the CRC for extraction. Additionally, since the generator polynomial $G(x) = x^{16} + x^{12} + x^5 + 1$ is the product of $(x + 1)$ and a primitive polynomial, the CRC can detect all parity errors, which are errors consisting of an odd-number of bit differences [10].

Based on previous work in CRC analysis, the CCITT-16 generator with an input of 256 bits has a Hamming Distance of 4 bits; precisely 6587 inputs with 4-bit-wide differences will be un-noticed [12]. Taking into account that a large amount of the input bits from SRAM are deterministic and that the majority of bytes only differ by a few values, a very small fraction of all the possible input sequences may not be detected. Additionally, there are no 5-bit errors that are undetectable [12]. Any 6-bit errors are highly unlikely to occur due to limited number of values each SRAM byte may contain.

The relevant properties of CRC-CCITT-16 as a mixing function are summarized as follows. (i) A primitive-based generator allows detection of all odd-numbered bit differences. (ii) Generator polynomial degree 15 guarantees detection of all 2-bit differences within the 256-bit input. (iii) A maximal-length LFSR design ensures pseudo-randomness in outputs. (iv) A small amount of 4-bit-wide input differences are undetectable, although wide bit differences are not critical to detect in this application. (v) All 5-bit-wide input differences are detectable. (vi) We don’t expect any 6-bit-wide input differences.

IV. CONCLUSION

We presented the design of a PRNG for commonly available embedded systems, ERHARD-RNG, that uses existing hardware peripherals as entropy sources. A previously design PRNG variant, Fortuna, was used along with three entropy sources: jitter in a low-power oscillator, measurements from an internal temperature sensor, and the start up state of SRAM. We provided entropy estimates for each source, and discussed how to build a system that efficiently uses memory and processing time resources to manage the collection and processing of sources. Furthermore, we proposed the use of a CRC hardware module to offload processing of the SRAM startup state to obtain an initial seed for the RNG. Using ERHARD-RNG as a reference, similar PRNGs can be implemented on other platforms to achieve reliable randomness, enabling security-critical functions in resource-restricted systems.

| User Input ID | User Input Time | Interrupt ID | Interrupt Time | Disk Access ID | Disk Access Time |
|---------------|----------------|--------------|----------------|---------------|-----------------|
| 0.7419        | 0.1147         | 0.3817       | 0.0075         | 0.3159        |                 |
| 2.0395        | 7.8972         | 0.9961       | 5.3211         | 5.2679        |                 |
| 1.3447        | 0.1147         | 0.7572       | 0.6692         | 0.9964        |                 |
| 1.8323        | 1.0752         | 0.9243       | 0.6000         | 0.8398        |                 |
| 0.7419        | 0.9996         | 0.3817       | 0.8555         | 0.3845        |                 |

**TABLE II: Linux source entropy estimates**
REFERENCES

[1] N. Ferguson and B. Schneier, Practical Cryptography (2nd Edition). New York, NY: John Wiley and Sons, 2003, pp. 1–432.

[2] A. Cherkaoui, V. Fischer, L. Fesquet, and A. Aubert, “A very high speed true random number generator with entropy assessment,” in Cryptographic Hardware and Embedded Systems - CHES 2013, G. Bertoni and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 179–196.

[3] S. Robson, B. Leung, and G. Gong, “Truly random number generator based on a ring oscillator utilizing last passagetime,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 61, no. 12, pp. 937–941, Dec 2014.

[4] D. E. Holcomb, W. P. Burleson, and K. Fu, “Power-up sram state as an identifying fingerprint and source of true random numbers.” IEEE Transactions on Computers, vol. 58, no. 9, pp. 1198–1210, Sept 2009.

[5] L. E. Bassham, III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, N. A. Heckert, J. F. Dray, and S. Vo, “Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications.” Gaithersburg, MD, United States, Tech. Rep., 2010.

[6] Statek Corporation, “An overview of oscillator jitter,” http://statek.com/wp-content/uploads/2018/03/mn35-Rev-B.pdf, 2007, online; accessed 20 February 2017.

[7] B. Sunar, W. J. Martin, and D. R. Stinson, “A provably secure true random number generator with built-in tolerance to active attacks.” IEEE Trans. Comput., vol. 56, no. 1, pp. 109–119, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1109/TC.2007.4

[8] MSP430F552x, MSP430F551x Mixed-Signal Microcontrollers datasheet, Texas Instruments, 11 2015, rev. M.

[9] Z. Gutterman, B. Pinkas, and T. Reinman, “Analysis of the linux random number generator,” in 2006 IEEE Symposium on Security and Privacy (S P’06), May 2006, pp. 15 pp.–385.

[10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, “Cyclic redundancy and other checksums,” in Numerical Recipes: The Art of Scientific Computing, 3rd ed. The Edinburgh Building, Cambridge CB2 8RU, UK: Cambridge University Press, 2007, ch. 22.4, pp. 1168–1175.

[11] C. Paar and J. Pelzl, “Shift register-based stream ciphers,” in Understanding Cryptography: A Textbook for Students and Practitioners. Springer-Verlag Berlin Heidelberg, 2010, ch. 2.3, pp. 41–49.

[12] P. Koopman, “Crc polynomial zoo,” http://users.ece.cmu.edu/~koopman/crc/crc16.html, Nov 2016. [Online]. Available: http://users.ece.cmu.edu/~koopman/crc/crc16.html