Cryptographic Randomness on a CC2538: a Case Study

Yan Yan
y.yan@bristol.ac.uk

Elisabeth Oswald
Elisabeth.Oswald@bristol.ac.uk

Theo Tryfonas
Theo.Tryfonas@bristol.ac.uk

University of Bristol,
UK
October 26, 2016

Abstract

Smart metering, smart parking, health, environment monitoring, and other applications drive the deployment of the so-called Internet of Things (IoT). Whilst cost and energy efficiency are the main factors that contribute to the popularity of commercial devices in the IoT domain, security features are increasingly desired. Security features typically guarantee authenticity of devices and/or data, as well as confidentiality of data in transit. Our study finds that whilst cryptographic algorithms for confidentiality and authenticity are supported in hardware on a popular class of devices, there is no adequate support for random number generation available. We show how to passively manipulate the on-board source for randomness, and thereby we can completely undermine the security provided by (otherwise) strong cryptographic algorithms, with devastating results.

1 Introduction

Applications for IoT flourish, leaving a great desire for not only energy efficient, cheap devices, but also for devices that support basic cryptographic functionality such as confidentiality and/or authenticity. Popular algorithms are e.g. the Advanced Encryption Standard[1] (AES) for confidentiality, and the Elliptic Curve Digital Signature Algorithm[2] (ECDSA) for authenticity, which, when used in conjunction, enable applications to establish secure end-to-end channels via e.g. Datagram TLS[3] (DTLS).

However whilst AES is a secure block cipher, one might require randomness to turn it into a secure encryption scheme for arbitrary length messages.

Somewhat similarly, ECDSA relies on a known-to-be-secure mathematical problem. However, it also requires large and securely generated random numbers. Consequently, when supporting cryptography the secure generation of random numbers is crucial.

In 2013, Texas Instruments (TI) launched a new System-on-chip (SoC), the CC2538[4], featuring secure channels over 802.15.4 via multiple cryptographic hardware accelerators. Partially because of these cryptographic accelerators, projects such as Contiki[5] and OpenWSN[6] began to support the CC2538 with enthusiasm. As of writing this paper, the chip features in the suggested list for Zigbee and 6LoWPAN solutions on TI's website[7].

However, despite all the cryptographic hardware support, the CC2538 does not have a Random Number Generator (RNG) dedicated for cryptographic applications; instead, the user's guide suggests to use a 16 bit Linear Feedback Shift Register (LFSR) as a Pseudo RNG (PRNG) where the seed is generated by the Radio Frequency (RF) module sampling from the radio noise. Whilst the user guide at no point suggested that this method should be used in conjunction with cryptographic algorithms, developers have little choice in the absence of alternatives. Also, in the absence of published attacks, there is often a temptation to ignore warnings towards insecure RNG implementations such as in [8].

1.1 Our Contribution

We show in this study that this choice proves catastrophic for cryptographic applications, not only because the in-built PRNG has only 16 bit entropy which can be easily predicted, but also because we are able to practically demonstrate how to bias the seed obtained from the RF module through the RF interface. Consequently, even if the weak in-built PRNG was replaced by a stronger component, the source for the seed could still be tampered with and thus render the system insecure. All the experimental work in this paper are performed on the latest Contiki release version 3.0.

Our paper is structured as follows. We begin in Section 1.3 with some Contiki RNG driver issues for CC2538. Section 2 revises why using a 16 bit LFSR as PRNG is a bad practice and we show how this design flaw can be exploited to break DTLS in Section 2.1, before reviewing the problem in Section 2.2. In Section 3 we explain how CC2538 samples the radio noise into random seed and then we demonstrate a bias attack in Section 4 which could be achieved given physical access to the device. Finally we conclude the paper in Section 5.

1.2 Related Work

The design flaw of using a 16 bit LFSR as PRNG has been reported by [8][9] on CC2430[10] and CC2530[11] respectively. These chips are the predecessors of CC2538 in the SimpleLink^{TM}series and they all adopted the same RNG design. The blogs reported the flaw and warned that it could easily be exploited to compromise the Z-Stack library[12] and Smart Energy Profile ECC in many

Smart Meter applications. We essentially 'rediscovered' that this poor design choice still features in the CC2538 product. However, whilst in previous work the possibility of injecting a jamming signal was contemplated, we are the first to actually examine the technical feasibility of this and to demonstrate a working attack.

1.3 Contiki Driver Issues

We made extensive use of Contiki in our research and fixed (and reported) some coding issues in the CC2538 RNG driver (Contiki release-3.0). These were, the reading out of the LFSR without ready check, a lack of validity check when reading random seed bits from the RF module, and a bug that drops the Most Significant Bit (MSB) and leaves the Least Significant Bit (LSB) to be constantly zero in the seed. We modified the code and fixed these issues in our experiments.

Another issue in the driver is that the CC2538 User's Guide[13] suggests only to use the lower byte (8 bits) as a random number but the driver actually used 16 bits in the LFSR. However, this coding mistake does not affect our result, as will be explained in later sections.

2 High Level PRNG Design and its Implication on DTLS Security

We begin this section by reviewing how the 'default' PRNG (as provided by the CC2538) turns a seed into a stream of pseudo random numbers using a 16-bit LFSR (see CC2538 User's Guide[13] Section 16.1). The polynomial that defines the feedback function is given as $x^{16} + x^{15} + x^2 + 1$ which corresponds to the well known CRC16 scheme[14]. When used as a PRNG (after it has been seeded), the input bit is simply set to zero. The CC2538 User's Guide[13] gives clear instructions on how to use the LFSR: by setting specific control bits to 01, the LFSR performs 13 CRC16 operations and its content can be read out as a random number.

Formally, because there are only 16 bits in the LFSR, we can denote the universal set of its possible values (or called states) $\mathbb{S}$ as:

$$\mathbb{S} = \{S_i | S_i \in \{2\}^{16}\} \tag{1}$$

Equation (1) implies that the LFSR can have no more than $|\mathbb{S}| = 2^{16} = 65536$ states.

We denote the LFSR update operation as:

$$F : \mathbb{S} \to \mathbb{S} \tag{2}$$

where F is 13 times of CRC16 operation on the current state according to the manual.

Denote the 16 bits random seed sampled from the radio noise as S^*. The PRNG output can be formalised as:

$$\begin{aligned} S_0 &= S^* \\ S_{i+1} &= F(S_i) \end{aligned} \tag{3}$$

Since $\mathbb{S}$ is finite and F is deterministic, the random number stream is cyclic. The longest non-repetitive PRNG output sequence under seed S^* can be represented as:

$$R_{S^*} = (F^0(S^*), F^1(S^*), ..., F^{n-2}(S^*), F^{n-1}(S^*)) \tag{4}$$

where $S^* = F^0(S^*) = F^n(S^*)$. Each call to the PRNG effectively returns the first element in the sequence and updates it by one cyclic left rotation. Since the elements within R_{S^*} are non-repetitive, we have $n \leq |\mathbb{S}|$ for any R_{S^*}, i.e. the cycle of PRNG output is at most 65536 calls.

For a re-sampled seed $S^{*'}$ inside R_{S^*}, i.e. $S^{*'} = F^k(S^*)$ where $k \in \mathbb{Z}_n$, the corresponding sequence $R_{S^{*'}}$ is:

$$\begin{aligned} R_{S^{*'}} =& (F^k(S^*), F^{k+1}(S^*), ..., F^{n-1}(S^*), F^0(S^*), F^1(S^*), ..., \\ & F^{k-2}(S^*), F^{k-1}(S^*)) \end{aligned} \tag{5}$$

Observing Equation (4) and Equation (5), we can see R_{S^*} is indeed $R_{S^{*'}}$ left rotated by $(n-k)$ times. This is equivalent to say that S^* generates identical output as $S^{*'}$ with $(n-k)$ preceding calls. As a result, assume consecutive PRNG calls on R_{S^*} returns a sequence of:

$$(S_i, S_{i+1}, ..., S_j)$$

Then the same sequence will eventually be replicated by calls on $R_{S^{*'}}$. This directly leads to the complete break of DTLS given the small space of $\mathbb{S}$, as we will explain in Section 2.1.

This property also indicates that any seed not in R_{S^*} generates a completely different sequence. By enumerating $\mathbb{S}$, we found there exists only four non-overlapping sequence for this CRC16 constructed PRNG, which are:

- R_{0x0001} with $n = 32767$.
- R_{0x0003} with $n = 32767$.
- R_{0x0000} with $n = 1$. $(F(0x0000) = 0x0000)$
- R_{0x8003} with $n = 1$. $(F(0x8003) = 0x8003)$

Notice that R_{0x0000} and R_{0x8003} are excluded in the driver due to their monadic output according to the manual[13]. The enumeration can be done on a CC2538 in less than a minute for such a small space of 65536.

2.1 Breaking DTLS

Contiki supports DTLS via an implementation called tinydtls[15]. Two cipher suites, namely Pre-Shared Key[16] (PSK) and ECDHE_ECDSA[17] are implemented by the latest available version (0.8.2) and the only supported curve is secp256r1[18]. In this paper we only discuss ECDHE_ECDSA.

Unfortunately, tinydtls does not implement its own RNG; instead it loops the Contiki API (random_rand()) which is then implemented by the CC2538 built-in PRNG (see tinydlts/dtls_prng.h) as we described in Section 2. As a result, the generated random numbers are from a very restricted set that is too small for any cryptographic use. This renders already any key generation vulnerable. A public Elliptic Curve (EC) key Q is the scalar multiple d of public base point G, i.e. $Q = [d]G$. Because d can only take 2^{16} values, it is trivial to build a table for a specific curve and public base point that contains all pairs of (d, Q). Consequently, upon observing a public EC key Q, one can use a simple table-lookup to deduce d. Due to the small entropy of CC2538 PRNG, such table contains only 65536 entries of (d, Q) pairs which can be computed and stored on a laptop within a few minutes.

Besides rendering key generation trivially insecure, one can further apply two trivial attacks during a DTLS handshake. As before, these attacks work easily because of the poor randomness and the fact that popular EC schemes use public, standardised base points.

ECDSA ECDSA[2] is an authentication scheme that allows a party to authenticate a message. In DTLS, it is used to sign the server parameters (details in [19]) during the handshake to provide server side authenticity. ECDSA requires a secret random number k to generate a point on the curve R via scalar multiplication of a base point. The x-coordinate r of this point becomes part of the signature. Hence it can be observed by the adversary, who can recover k by searching r in the look up table of pre-computed points. He can then recover the secret signing key d by computing:

$$\begin{aligned} e &= SHA-1(m) \\ d &= r^{-1}(sk - e) \mod n \end{aligned} \tag{6}$$

ECDHE ECDHE[17] is a key exchange protocol that allows two party to derive a shared secret. In DTLS, ECDHE is performed at the end of DTLS handshake to derive a shared secret, which is then used to derive the symmetric session key for application data encryption. ECDHE essentially performs a Diffie-Hellman key agreement, i.e. one party computes $Q_A = [r_A]G$, the other party independently computes $Q_B = [r_B]G$; both parties exchange points, and so are able to derive $Q_{AB} = [r_A][r_B]G = [r_A]Q_B = [r_B]Q_A$. Because G is public, it is again possible to derive r_A and r_B by looking up Q_A and Q_B in the pre-computed table. Once these quantities are known to the adversaries, they can also compute Q_{AB} and hence the session key.

```
static unsigned long seed = 1;

int
rand (void)
{
  return do_rand (&rand);
}
```

Figure 1: rand() implementations in stdlib

We have tested the attacks by sniffing two CC2538 nodes performing handshake using the example code provided in tinydtls. The secret keys have been successfully recovered using the look up table we generated.

2.2 (P)RNG implementations in Contiki

Investigating (P)RNG implementations in other platforms supported by Contiki, we realised that most of them do not have dedicated PRNG implementations and by default wrap rand() in stdlib as their PRNG. We traced some of the open sourced stdlib implementations. For the majority of the libraries, i.e. stdlib for ARM[20], AVR[21] and MSP430[22], the rand() implementation can be abstracted as Figure 1. The type of variable *seed* may vary on different platforms. The *do_rand()* function outputs a congruent of linear transformation of *seed* and updates *seed* by the output.

It is clear that such design would also yield into a predictable random number stream with cycle no longer than the range of *seed*, as the same *seed* returns the same output. On the above platforms, the cycles are no longer than 2^{32}, 2^{16} and 2^{16} calls respectively.

As a straightforward improvement, we suggest using more sophisticated PRNG implementation for cryptographic applications. [23] has recommended several PRNG constructions based on approved hash functions and block ciphers. Specifically for CC2538, SHA-256 and AES have hardware coprocessor support and therefore can be considered candidates for implementing cryptographically secure PRNG according to [23].

3 Using RF Noise as True Random Number Generator

PRNGs require an unpredictable seed, i.e. a true random number as a starting point. Higher end devices, such as security ICs hence come with a dedicated true random number generator. The CC2538 manual does not claim that using RF noise is a suitable source for random numbers in a cryptographic context, however, in the absence of another source, developers are bound to use what is available. The CC2538 User's Guide[13] explains to fill the SOC_ADC_RNDL register with random bits from the Intermediate Frequency Analogue-to-Digital

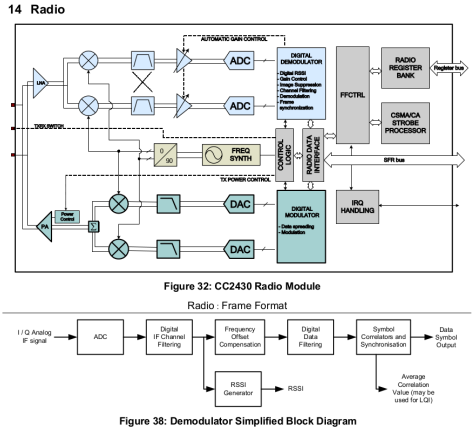


Figure 2: CC2430 RF Design, from CC2430 user manual[10]

Converter (IF_ADC) in the RF receive I/Q channels to seed the PRNG (Section 16.2.2). The user guide[13] also reports on the good quality of the randomness (Section 23.12).

To verify the claims in the manual, we applied the NIST Statistical Test Suite[24] on 13263600 bits sampled by this seeding method. Since one bit is returned upon each read to the RNG register, we concatenated all bits into one bit stream. The bits passed all tests in the NIST test suite, with $P(0) = 0.49995001$ and $P(1) = 0.50004999$, which shows that the RF noise (when not tampered with) is indeed a good source for random numbers. However, it remains unclear whether such source can practically be influenced by crafted RF signals.

3.1 Reverse Engineering the TRNG Design

The documents supplied by TI do not explain further details of how IF_ADC in the receive I/Q channels are translated to random bits. We have neither been able to find any open document describing the RF design of CC2538. However, we noticed that the same design has been applied to several products in TI's SimpleLink^{TM}series, some of which provided a better explanation of their RF core and RNG designs.

In the CC2430 user manual[10], we found a description of its RF core as in Figure 2 which explains that the input analogue signal to IF_ADC goes through the following components:

- Low Noise Amplifier (LNA) which amplifies the signal.
- Mixer which down converts the signal frequency. The Frequency Synthesiser is used as the local oscillator.
- Band pass filter which removes the out of band signals.

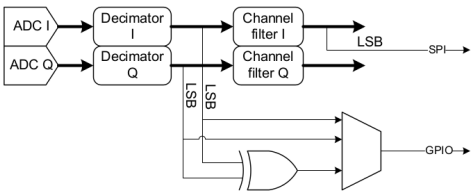


Figure 3: CC2520 RNG Design, from CC2520 user manual[25]

- The Automatic Gain Control (AGC) circuit further adjusts the signal strength to the input level of ADC.

The CC2520 Data Sheet[25] explains the random bit is actually the LSB from ADC output: (Section 24 in CC2520 Data Sheet[25])

> Single random bits from either the I or Q channel (configurable) can be output on GPIO pins at a rate of 8MHz. One can also select to xor the I and Q bits before they are output on a GPIO pin. These bits are taken from the least significant bit in the I and/or Q channel after the decimation filter in the demodulator.

A block diagram is also provided, as shown in Figure 3.

Interestingly, we noticed that CC2538, CC2520, CC253X and CC2540/41 reported exactly the identical randomness test result in their user manuals ([13] [25] [11]). We suspect the above evidence showed that CC2538 is very likely to have adopted the same designs.

The designs above explained the nice randomness of the seeding method. Denote V_s as the analogue RF signal and N as the noise, the analogue input to the ADC, denote as V_{in}, can be represented as:

$$V_{in} = V_s + N \tag{7}$$

The noise N can be induced by multiple sources in practice, including noise produced by the signal source, environmental noise, and noise induced by the components in the device itself, etc.

The random bit b can therefore be represented as:

$$b = LSB(V_{in}) = LSB(V_s + N) \tag{8}$$

where $LSB() \in \{0, 1\}$ represents the operation of taking the LSB of A/D conversion output.

From Equation (8) we observe that any difference in V_{in} that is larger than the scale of ADC, i.e. the voltage represented by the LSB of ADC, could flip b. According to the CC2538 Datasheet[26], the receiver can be sensitive to signals down to $-97dBm$ (typical value with $T_A = 25°C$, $V_{DD} = 3V$ and $f_C = 2440MHz$). On the other hand, the typical environmental noise in our

experimental environment is about $-92dBm$ which is significantly higher than the receiver sensitivity. We consider this reading as a generic office use case and the result of randomness test as evidence to the sampling method.

4 Biasing the RF Signal in Practice

Equation (8) indicates that the random bit b is jointly determined by the signal V_s and noise N. Although an adversary can generate arbitrary signals, i.e. V_s is fully controlled by the adversary, it is clear that controlling N is difficult in practice. For instance, noises accumulated by different amplification stages are physically inevitable and intrinsic to the physical device. Hence it is not straightforward to fully control $V_{in} = V_s + N$ in practice.

An alternative attempt is to provide the RF with an 'illegal' input V_{in}. We considered two methods in our experiments: saturation and decimation. Saturation attempts to provide the RF with a strong signal that is above its acceptance level, whereas decimation attempts to suppress any RF signal to beneath the receiver sensitivity.

Ideally we expect these illegal inputs will trigger the ADC into a fault state which could potentially results into a predictable ADC output and thus biased b. But in practice, decimation does not seem practical for the same reason that noise induced by the circuit itself is physically inevitable. This made saturation the only viable option for us. We further note that the undisclosed circuit design of the device also posed a difficulty in our experiments. Without knowledge of the exact circuit design, we had to perform black box experiments.

4.1 Concrete experiments

We used OpenMote[27], a CC2538 powered SoC, for our experiments. We extended the length of each seed from RF to 128 bits in our experiments in coping to a potential PRNG design based on AES-128, although we still consider the bits are generated bitwise when applying the NIST test suite.

4.2 Strong sine wave signal

The first signal we attempted was a strong sine wave signal. According to CC2538 data sheet[26], the saturation signal strength for the RF receiver is $10dBm$. We have attempted to increase the input signal strength up to $13dBm$, which is roughly double of the saturation voltage, but no bias was observed. The seed sampled under this signal has passed all tests in the NIST test suite.

The result implies that the AGC circuit could have tuned down the signal which might have consequently prevented the seed from being biased. Although the exact AGC design for CC2538 is unclear, Figure 4 demonstrates an example of AGC design using 4 Voltage Controlled Amplifiers (VGAs). The output signal is parallelly connected to a detector to estimate the signal strength. The output of detector is compared to a reference voltage and their difference is provided as

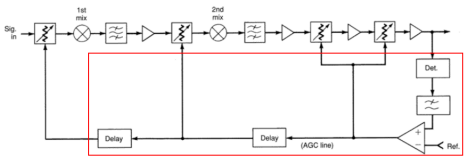


Figure 4: Example of receiver AGC, from [29]

Table 1: GRC Signal Source Configuration

Sample Rate	8 MHz
Output Type	Complex
Waveform	Constant
Amplitude	0
Offset	1
IF Gain	[0, 30] dB
Output Voltage Amplitude	[0,176.0] mV

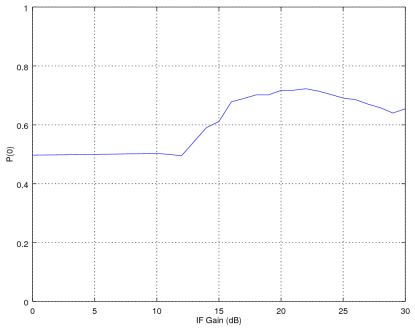
a feedback to adjust the control voltage of VGAs. To prevent signal distortion caused by abrupt voltage change, such as during a lightning storm, many AGC design adopts an attack time (or called settle time) before it adjusts the gain. [28] provides a detailed description of different AGC designs.

4.3 Strong constant signal

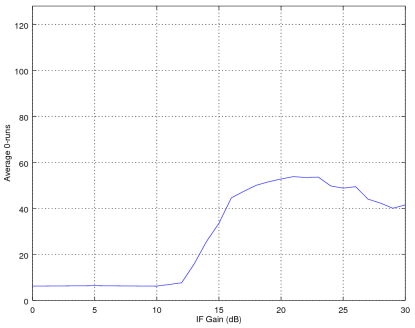
We then attempted a strong constant signal. The idea is to treat the whole circuit as a deterministic compression function that maps any V_{in} to $\{0, 1\}$. Under this assumption, the same V_{in} should always generate the same b, either 0 or 1. In order to achieve constant V_{in} in Equation (7), V_s needs to be significantly greater than N to suppress its impact in Equation (8), as any ADC would have only a limited resolution.

For experimental purpose, we have configured three programmable LNAs in the AGC to their maximum gain ($6 + 21 + 9 = 36(dB)$) and has disabled the attenuator in Anti Aliasing Filter (AAF, up to $9dB$). We consider these modifications can be compensated by a strong signal amplifier in practice. The signal source is implemented by Gnu Radio Companion (GRC)[30] with HackRF One[31], connected to the target OpenMote through a SMA cable for the best signal strength. Table 1 lists the configuration which effectively generates a carrier wave on desired frequency.

Applying the signal, we observed abnormal 0-runs, i.e. consecutive 0 bits, appeared in the seeds as we increase IF gain to values above $10dB$. Figure 5a shows how $P(0)$ is biased and Figure 5b shows the average number of bits of longest 0-runs in each seed. We can see that the bias has reached its peak at IF_Gain $= 22dB$ in both figures. At such gain 27.709% of the 128 bit seeds have longest 0-runs over 64 bits. It is not a surprise to see the sampled seeds have



(a) $P(0)$ to IF Gain

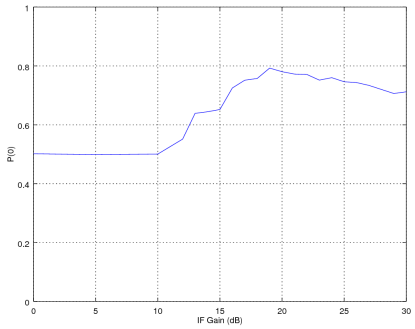


(b) Average longest 0-runs in each seed to IF Gain

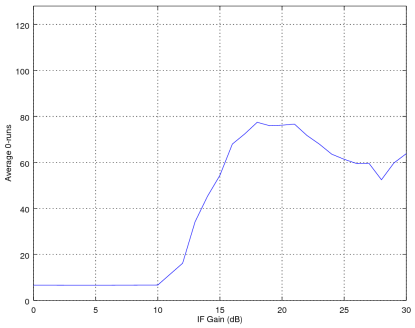
Figure 5: Biased Seed on OpenMote. Signal source amplitude from 19.5mV (10dB), 76.0mV(22dB) to 176.0mV (30dB).

failed nearly all tests in the NIST test suite, indicating they have been strongly biased by our signal. We cannot determine the exact cause of bias decrease for IF Gain over 22dB due to lack of circuit design, but one potential caused might be the distortion under strong signal strength.

We also re-applied the signal to the same OpenMote we previously used in the strong sine wave signal experiments. A even stronger bias is observed as shown in Figure 6, with 17.820% of the seeds ended in 128 consecutive 0 bits. This may be caused by the strong sine wave signal in the previous experiments which permanently biased the device. We therefore restored its AGC configuration to default and re-ran the NIST test suite but the sampled seed passed all tests as before. We also tested using example applications provided by Contiki and found no malfunctioning on the device. The permanent bias does not seem to affect the device under normal operational status and can only be triggered by the constant signal. This leads to a very dangerous attack where devices could be primed in such a way that they remain functional under normal oper-



(a) $P(0)$ to IF Gain



(b) Average longest 0-runs in each seed to IF Gain

Figure 6: Biased Seed on OpenMote used in previous experiments. Signal source amplitude from 19.5mV (10dB), 76.0mV(22dB) to 176.0mV (30dB).

ating conditions, and eventually 'activated' via supplying the activation signal upon which they are unable to produce random numbers and hence all DTLS connections would be completely insecure. We were able to replicate this attack on brand new devices with factory settings.

5 Conclusion

In this paper we reviewed the provision for cryptographic random numbers on the CC2538 and related devices. First, we discussed the poor choice of using a 16 bit LFSR as PRNG and demonstrated how this design flaw can be exploited to break DTLS running on these devices. We also found that the provision for randomness within the popular Contiki software and DTLS implementation tinydtls is inadequate. Any open source efforts, or indeed also any products, that built on them should review their instantiation of random numbers carefully.

Secondly we investigated how to tamper with the RF source and showed in practice how to configure signals to that end. We reverse engineered the design of the path that produces random bits from the RF module, and developed some attacks that can bias the random bits in practice. This shows that even if the poor PRNG was replaced with a sound one, the source for the seed of any PRNG on the CC2538 is vulnerable to practical attacks. However, the signal strength required for the attack might not be achievable unless the adversary have direct physical access to the device.

We believe that the same design choices have also been adopted by many other products in the CC series including CC2420[32], CC2430[10], CC2520[25] and CC253X, CC2540/41 series[11]. To the best of our knowledge all these products suffer from the same problems. Only the latest CC26XX/CC13XX[33] series has abandoned this design and implemented a dedicated RNG suitable for cryptographic purposes. We recommend to update the legacy devices for security sensitive application.

6 Acknowledgement

We have many thanks to (alphabetically) George Oikonomou for providing us much help in Contiki OS and the OpenMote devices, Geoff Hilton who helped us on RF designs and Jake Longo Galea who offered many signal processing advises.

7 Related Source Code

Source code related to this paper can be found at [34]. The repository contains all source file to reproduce the experiments done in this paper, including:

- **prngtest**: Contiki application that iterates the CC2538 PRNG.
- **genr.py** and **secp256r1mult**: Tools to generate the EC key pair lookup table for CC2538 PRNG.
- **cc2538seed**: Contiki application that samples the RF seed for CC2538.
- **biasseed.py**: Implementation of the strong constant signal for HackRF One.

Detail usage documented in readme.txt.

References

[1] P. FIPS, "197, advanced encryption standard (aes), national institute of standards and technology, us department of commerce (november 2001)," *Link in: http://csrc. nist. gov/publications/fips/fips197/fips-197. pdf.*

[2] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[3] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security," RFC 4347 (Proposed Standard), Internet Engineering Task Force, Apr. 2006, obsoleted by RFC 6347, updated by RFC 5746. [Online]. Available: http://www.ietf.org/rfc/rfc4347.txt

[4] [Online]. Available: http://www.ti.com/product/CC2538/description

[5] [Online]. Available: http://http://www.contiki-os.org/

[6] [Online]. Available: https://openwsn.atlassian.net/wiki/pages/viewpage.action?pageId=688187

[7] [Online]. Available: http://www.ti.com/lsds/ti/wireless_connectivity/zigbee/products.page#

[8] [Online]. Available: https://rdist.root.org/2010/01/11/smart-meter-crypto-flaw-worse-than-thought/

[9] [Online]. Available: http://travisgoodspeed.blogspot.co.uk/2009/12/prng-vulnerability-of-z-stack-zigbee.html

[10] [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2430.pdf

[11] [Online]. Available: http://www.ti.com/lit/ug/swru191f/swru191f.pdf

[12] [Online]. Available: http://www.ti.com/tool/z-stack

[13] [Online]. Available: http://www.ti.com/lit/ug/swru319c/swru319c.pdf

[14] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[15] [Online]. Available: https://sourceforge.net/projects/tinydtls/

[16] P. Eronen and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)," RFC 4279 (Proposed Standard), Internet Engineering Task Force, Dec. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4279.txt

[17] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)," RFC 4492 (Informational), Internet Engineering Task Force, May 2006, updated by RFCs 5246, 7027. [Online]. Available: http://www.ietf.org/rfc/rfc4492.txt

[18] S. SEC, "2: Recommended elliptic curve domain parameters," *Standards for Efficient Cryptography Group, Certicom Corp*, 2000.

[19] L. Bassham, W. Polk, and R. Housley, "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 3279 (Proposed Standard), Internet Engineering Task Force, Apr. 2002, updated by RFCs 4055, 4491, 5480, 5758. [Online]. Available: http://www.ietf.org/rfc/rfc3279.txt

[20] [Online]. Available: https://chromium.googlesource.com/native_client/nacl-newlib/+/master/newlib/libc/stdlib/rand_r.c

[21] [Online]. Available: https://github.com/vancegroup-mirrors/avr-libc/blob/master/avr-libc/libc/stdlib/rand.c

[22] [Online]. Available: https://sourceforge.net/p/mspgcc/gcc/ci/master/tree/libiberty/random.c

[23] E. B. Barker and J. M. Kelsey, *Recommendation for random number generation using deterministic random bit generators (revised)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, 2007.

[24] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," DTIC Document, Tech. Rep., 2001.

[25] [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2520.pdf

[26] [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2538.pdf

[27] [Online]. Available: http://www.openmote.com/

[28] J. P. A. Pérez, S. C. Pueyo, and B. C. López, *Automatic Gain Control*. Springer, 2011.

[29] I. Rosu, "Automatic gain control(agc) in receivers." [Online]. Available: http://www.qsl.net/va3iul/Files/Automatic_Gain_Control.pdf

[30] [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion

[31] [Online]. Available: https://greatscottgadgets.com/hackrf/

[32] [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2420.pdf

[33] [Online]. Available: http://www.ti.com.cn/cn/lit/ug/swcu117d/swcu117d.pdf

[34] [Online]. Available: https://github.com/Salties/MyRepository/tree/master/experiments/cc2538rng