Secure PRNG Seeding on Commercial Off-the-Shelf Microcontrollers

Anthony Van Herrewege[1], Vincent van der Leest[2], André Schaller[3], Stefan Katzenbeisser[3], and Ingrid Verbauwhede[1]

[1] KU Leuven Dept. Electrical Engineering-ESAT/SCD-COSIC and iMinds
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
firstname.lastname@esat.kuleuven.be
[2] Intrinsic-ID, Eindhoven, The Netherlands
http://www.intrinsic-id.com
[3] Security Engineering Group, Technische Universität Darmstadt and CASED, Germany
lastname@seceng.informatik.tu-darmstadt.de

Abstract. The generation of high quality random numbers is crucial to many cryptographic applications, including cryptographic protocols, secret of keys, nonces or salts. Their values must contain enough randomness to be unpredictable to attackers. Pseudo-random number generators require initial data with high entropy as a seed to produce a large stream of high quality random data. Yet, despite the importance of randomness, proper high quality random number generation is often ignored. Primarily embedded devices often suffer from weak random number generators. In this work, we focus on identifying and evaluating SRAM in commercial off-the-shelf microcontrollers as an entropy source for PRNG seeding. We measure and evaluate the SRAM start-up patterns of two popular types of microcontrollers, a STMicroelectronics STM32F100R8 and a Microchip PIC16F1825. We also present an efficient software-only architecture for secure PRNG seeding. After analyzing over 1 000 000 measurements in total, we conclude that of these two devices, the PIC16F1825 cannot be used to securely seed a PRNG. The STM32F100R8, however, has the ability to generate very strong seeds from the noise in its SRAM start-up pattern. These seeds can then be used to ensure a PRNG generates high quality data.

Keywords: Secure seeds, PRNG, Static RAM, Security

1 Introduction

The generation of high quality random numbers is crucial to many cryptographic applications. Almost every cryptographic protocol involves the use of keys, nonces or salts which are unpredictable to attackers. Such values must exhibit a sufficient degree of randomness, i.e., contain enough entropy. In addition, re-keying is applied regularly to prevent a wear-out effect of secret keys. Finally, public-key cryptosystems, such as RSA and ElGamal, rely on random numbers to generate public/private key pairs.

Yet, despite its importance, proper high quality random number generation is often ignored. Random data with too little entropy results in weak keys, nonces or salts, which can then be guessed with minimal effort, thereby compromising even the strongest

cryptosystem. Hence, the quality of random data ultimately affects the level of security of cryptographic primitives and protocols in practice. Although many cryptographically secure pseudo-random number generators (PRNG) exist, all of them require to be seeded with initial data containing sufficient entropy. Once seeded, they are able to generate high quality random output for long periods of time. Providing PRNGs with a low quality initial seed, however, will cause them to generate weak, predictable output.

Neglecting to ensure sufficient entropy in PRNG seeds gave rise to several security incidents. A famous case was the OpenSSL implementation in Debian [15]: by accidentally decreasing the number of available randomness sources for seeding, the generated random numbers became predictable. This incident affected numerous TLS/SSL connections, keys for SSH accounts, as well as the security of Tor users [4]. More recently, Heninger et al. [7] and Lenstra et al. [11] conducted an Internet-wide survey and looked for security problems in public keys and certificates of TLS and SSH servers, caused by low quality random number generation. The authors were able to recover private keys of several devices due to common factors in public RSA keys. Their results indicate that primarily embedded devices, such as routers, firewalls and VPN appliances, are affected. The source of these problems are likely PRNGs that were not seeded with high entropy data on start-up.

In this work, we focus on identifying and qualifying Static Random Access Memory (SRAM) in commercial off-the-shelf (COTS) microcontrollers as an entropy source for PRNG seeding. We take advantage of the fact that the start-up values of SRAM are noisy. This noise is collected upon boot time to derive a high quality, high entropy PRNG seed by applying a hash function to the initial memory contents. We measure and evaluate the entropy in SRAM start-up patterns in two common types of microcontrollers, an STMicroelectronics STM32F100R8 (ARM Cortex-M3) and a Microchip PIC16F1825. Furthermore, we suggest an architecture for seed extraction and pseudo-random number generation, which makes efficient use of the available resources in a COTS microcontroller.

The paper is structured as follows. After surveying related work in Section 2, we analyze and evaluate the noise in the SRAM start-up patterns of the aforementioned microcontrollers in Section 3. In Section 4, we present the architecture for an efficient SRAM-based secure seed generator and PRNG. Furthermore, we present our attacker model and discuss practical aspects relating to the implementation of our architecture. Finally, we conclude the paper in Section 5.

2 Related Work

Random number generation. In order to generate random numbers for cryptographic applications on microcontrollers, two basic methods can be used. The first method requires a physical source, which is truly random and from which bits can be derived directly. Such non-deterministic sources derive their randomness from underlying physical properties that exhibit unpredictable behaviour. Examples of such sources of randomness in chips are free running oscillators connected to a shift register [19] and noise on the lowest bits of AD converters [17], but many more exist. There are two important downsides to most of these physical RNG constructions. Firstly, they require specific hardware to extract the randomness from the physical entities on the device. Secondly,

the throughput of such RNGs is generally relatively low. This is problematic when large streams of random bits are required for cryptographic applications.

The second approach to generate randomness is by using PRNGs, which are deterministic algorithms. An introduction to PRNGs can be found in [1]. The output of such a generator only seems random to observers without prior knowledge. However, if an observer knows which data has been used as a seed for the PRNG, then he will be able to calculate all output values of the generator. Hence, this seed value should be randomly chosen (and kept secret). The upside of this type of generator is that it can be implemented completely in software and therefore does not require any hardware additions to a microcontroller. Also, it can produce a stream of (pseudo-)random output bits at a high throughput rate.

The benefits of a PRNG greatly outweigh those of a true random number generator on a COTS microcontroller. The necessity of generating a truly random seed is of crucial importance though. Our goal is to identify and evaluate methods that can be used to generate strong seeds for a PRNG and that are already available in COTS microcontrollers, thus requiring no hardware modifications.

SRAM as sources of entropy. Our approach of generating a seed value is based on random noise extracted from the power-up state of SRAM modules, which are part of COTS microcontrollers. SRAM bit cells are designed as cross-coupled inverters, which exhibit a bi-stable behaviour. When powered on these cells eventually settle from a meta-stable state to a stable state, either zero or one. It was shown by Guajardo et al. [6] that memory cells are often biased to zero or one, due to uncontrollable physical conditions during the manufacturing stage leading to one of the inverters being slightly stronger than the other. Some cells, however, will be almost perfectly symmetric, which leads them to settle to an unpredictable value at start-up. It is the noise due to these cells which we exploit in order to generate high quality random seeds.

The general idea of using SRAM as a source for PRNG seeds was investigated in [8] as well as in [10]. However, the former paper proposes to use a universal hash to generate a single random number at start-up. This technique is then verified on an external SRAM module. However, it is not investigated whether the approach works on the embedded SRAM in COTS microcontrollers. In the latter paper, the feasibility of creating a strong PRNG with the use of random data from an ASIC containing SRAM-based Physically Unclonable Functions (PUFs) [12] is investigated. In contrast, our goal is to identify COTS microcontrollers which can be used without any hardware modifications to support high quality random number generation and hence cryptographic protocols.

In Mowery et al. [18], the authors gather entropy from the clock jitter between different clock domains on a CPU. Their approach is quite slow, however, and obviously does not work on embedded devices with only a single clock domain. In cases where their approach is feasible, it can be combined with our method in order to increase the amount of gathered entropy.

3 Evaluation of Entropy in SRAM Start-up Values

For the purpose of extracting a random seed from SRAM start-up values, it is important to investigate their entropy contents. In this section, our approach to quantify the

entropy quality of the SRAM patterns (namely the calculation of min-entropy) is explained. We will also present the hard- and firmware used to measure the SRAM start-up pattern of two popular COTS microcontrollers. Finally, we show and discuss the measurements for these two devices under different ambient conditions.

The first investigated microcontroller is the 32-bit STM32F100R8 by STMicroelectronics, an ARM Cortex-M3 chip. The second one is the 8-bit PIC16F1825 by Microchip, part of Microchip's range of high-end 8-bit processors. Both of these chips were chosen for their popularity. The STMicroelectronics chip was chosen due to the Cortex-M family being ARM's fastest licensing processor family to date. The Cortex-M family was licensed 168 times by Q4 2012 [13], with 23 billion of these chips sold last year. Microchip has the 4th largest market share in the extremely fractioned microcontroller market [14].

3.1 Method of deriving min-entropy

To extract a high quality seed from the SRAM start-up values we have to examine their randomness properties in terms of entropy. In particular, the amount of entropy must be present in the noise of SRAM start-up patterns should be determined. For this purpose we will be calculating the min-entropy in the same manner as was done in [10]. This method is based on the NIST specification [3] that defines min-entropy as the worst-case (i.e., the greatest lower bound) measure of uncertainty for a random variable.

For a binary source, we can define the min-entropy as

$$H_{min} = -\log_2(\max(p_0, p_1)),$$

where p_0 and p_1 are the probabilities of 0 and 1 occurring. Assuming that all bits from the SRAM start-up pattern are independent, each bit i can be viewed as an individual binary source. For each of these sources we estimate the probabilities p_0^i and p_1^i of powering up in state 0 or 1, by repeatedly measuring the power-up values of the SRAM. In case m subsequent measurements are performed, p_0^i denotes the number of occurrences of a zero, divided by m and $p_1^i = 1 - p_0^i$. For n independent sources (where n is the length of the start-up pattern), we have:

$$H_{min} = \sum_{i=1}^{n} -\log_2(\max(p_0^i, p_1^i))).$$

Hence, under the assumption that all bits are independent, we can sum the entropy of each individual bit to derive the min-entropy of the entire SRAM. In the remainder of this work, we generally denote the available min-entropy as a percentage of the total available SRAM size.

3.2 Measurement setup

In this subsection, we present the soft- and hardware setup used to evaluate COTS microcontrollers. First, we present the functionality and requirements of the firmware that has to be put into each microcontroller to be measured. Thereafter, we describe the hardware construction used to extract start-up values for later evaluation.

Firmware design Every microcontroller to be measured should be programmed with firmware that, on power-up, initializes the serial port and then starts transmitting the value of each SRAM byte in sequence. Once finished, it should enter an idle loop. Care should be taken not to use any of the SRAM storage while doing this. Most microcontrollers have a several working registers to store variables, such as a pointer to the current SRAM byte, and thus this will be easy to achieve. However, some microcontrollers, such as the Microchip PIC16 family, only have a single working register and therefore, in order not to write data to any SRAM byte, some variables will have to be stored in unused configuration registers.

Hardware setup. To get some initial measurements of the SRAM power-up patterns, we first conduct our experiments manually. In this setup, the microcontroller to be measured has its power lines and serial port connected to an external serial TTL–to–USB converter. The converter is connected to a self-powered USB hub. After an SRAM measurement has been taken, power to the microcontroller is switched off (i.e. left floating) for at least 10 seconds. This is to ensure that the microcontroller has discharged completely and that the SRAM will contain fresh data on the next power-up. Although this discharging cycle works fine for the STM32F100R8 devices, it does not for the PIC16F1825 devices, which keep their SRAM values for over 10 minutes when their supply pins are left floating.

In order to extract start-up patterns faster and efficiently, we created a custom measurement board. The requirements for this board are:

1. Allow connection of many microcontrollers at once.
2. Be extensible with regards to number of attached microcontrollers.
3. Support remote setup.
4. Make automated, unsupervised measurements possible.
5. Support any realistic baud rate.
6. Support any arbitrary SRAM size.
7. Supply upwards-going, fast rising ($\leq$2 ms) Vcc signals.
8. Actively discharge microcontrollers that are not being measured.

Requirements 1 and 2 are satisfied by using (de)multiplexers for the power supply and serial transmission (TX) lines of the attached microcontrollers. The controller board interfaces with a PC, thereby meeting requirements 3 and 4. The controller clock signal is generated with a specialized clock, and the baud rate can also be set though the PC interface, thus fulfilling requirement 5. Requirement 6 is met by detecting when the TX line of the currently powered microcontroller goes idle, at which point the controller board advances to the next connected microcontroller. In order to generate realistic start-up patterns, requirement 7 should be met. We used an oscilloscope to verify that this was the case for our controller board. Finally, requirement 8 is necessary in order to erase the state of the SRAM completely on power-down. The demultiplexer on our controller board connects non-active power lines to ground, thereby this last requirement is met as well.

A simplified schematic of our design is shown in Fig. 1. In its current state, it allows us to connect up to 16 microcontrollers. This can be extended to at least 1024 devices, in case this should prove necessary.

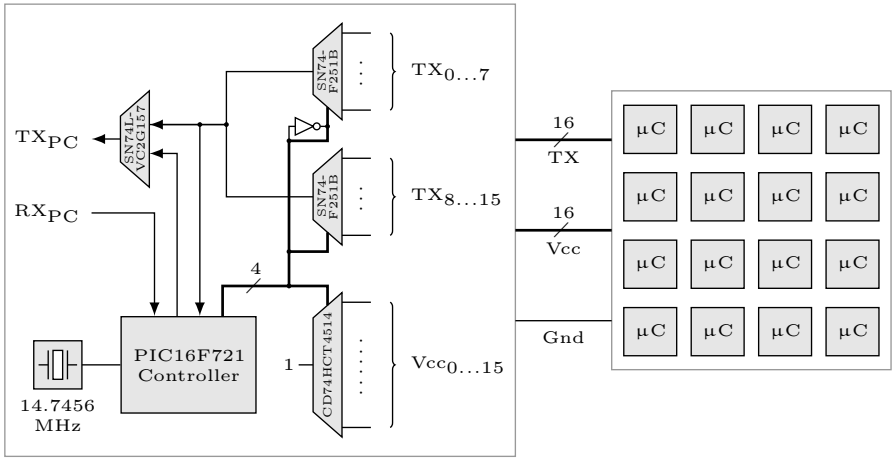

Fig. 1. High-level schematic of the measurement controller board (*left*) with a board of microcontrollers to be measured attached (*right*).

3.3 STMicroelectronics STM32F100R8

The first device that has been tested for entropy in the SRAM start-up values is the STM32F100R8 from STMicroelectronics. This is a 32-bit ARM Cortex-M3 device with 8 KiB of SRAM. Of this device 10 samples have been used to perform a large number of measurements. An example of a start-up pattern (measured at +25 °C) of the SRAM in this microcontroller can be found in Fig. 2.

To make sure that the STM32F100R8 provides sufficient noise entropy under different circumstances, measurements have been performed at −30°C, +25°C and +90°C. Using these measurements the min-entropy has been derived (with the method described in Section 3.1), the results[4] for the different conditions can be found in Table 1.

Table 1. Min-entropy results for STM32F100R8 SRAMs at different temperatures. Min-entropy denoted as percentage of total available SRAM.

Temp.	Microcontroller ID									
[°C]	1	2	3	4	5	6	7	8	9	10
−30	5.3%	5.3%	5.4%	5.5%	5.4%	5.3%	5.4%	5.2%	5.3%	5.8%
+25	6.6%	6.6%	6.7%	6.8%	6.7%	6.5%	6.8%	6.5%	6.7%	6.7%
+90	6.3%	6.5%	6.5%	6.6%	6.2%	6.5%	6.5%	6.2%	6.5%	6.5%

From the results in Table 1 it is clear that the STM32F100R8 devices contain a minimum amount of 5.2% min-entropy under all tested circumstances. Given the

[4] Min-entropy is expressed here as a percentage of the length of the start-up pattern. Hence, 6.0% in this 8 KiB memory is approximately equal to 491.5 bytes min-entropy.

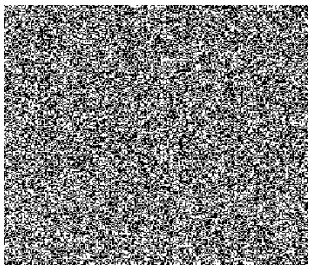

Fig. 2. Example start-up pattern of STM32F100R8 (8 KiB SRAM) at +25 °C. White represents a bit with value 0, black a bit with value 1.

fact that the measured SRAMs have a size of 8KiB, it is evident that by using these memories as input for a hash function it is no problem to derive a truly random seed for a PRNG[5]. If for example, assuming the entropy is evenly spread out over the entire SRAM, we would like to derive a truly random seed of 256 bits and consider a min-entropy of 3% (which is on the safe side, given the lowest min-entropy of 5.2% from the analysis), the required amount of SRAM to derive this seed is only 1.04KiB. For a more cautious approach, in which no assumptions are made about the entropy distribution, see Appendix B.

3.4 Microchip PIC16F1825

The second commercially available microcontroller that has been evaluated, is the Microchip PIC16F1825. This is an 8-bit microcontroller with 1 KiB of SRAM. Under the same conditions as described in the previous section, a large number of measurements of the SRAM start-up patterns of 16 different devices have been performed. A plot of one of these start-up patterns (measured at +25 °C) is given in Fig. 3. It is evident from this plot that there is severe biasing in the start-up pattern. The plot clearly shows that the bits from the PIC16F1825 memories possess a pattern which is far from random. To be more precise: the bits of every alternating byte have a preference to start-up either as a 0 or a 1. A pattern as can be seen in Fig. 3 is present in every PIC16F1825 device measured. The preference towards 0 or 1 for each byte results in a lower noise entropy, because it is less common for these bits to flip since they have a preferred state to start in. Using these measurements, the min-entropy of the SRAM noise has been determined in the same way as for the STM32F100R8 devices. The resulting min-entropies at different temperatures can be found in Table 2.

In comparison to the results of the STM32F100R8 devices, it is clear that the noise entropy for the PIC16F1825 is significantly lower. For the measurements at room and high temperatures this can be explained by the severe biasing of the start-up pattern, which has been discussed already. For the low temperature (at which the min-entropy is

[5] A less extensive evaluation STM32F051R8 and STM32F100RB devices seems to suggest that other devices in the STM32 family contain an equally high amount of entropy in their SRAM start-up patterns.

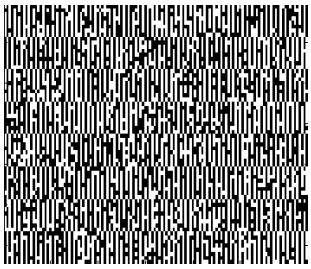

Fig. 3. Example start-up pattern of PIC16F1825 (1 KiB SRAM) at +25 °C. White represents a bit with value 0, black a bit with value 1.

Table 2. Min-entropy results for PIC16F1825 SRAMs at different temperatures. Min-entropy denoted as percentage of total available SRAM.

Temp.	Microcontroller ID							
[°C]	1	2	3	4	5	6	7	8
−30	0.9%	1.0%	0.3%	0.2%	0.5%	0.2%	0.2%	0.2%
+25	1.9%	2.0%	1.8%	1.8%	1.9%	1.9%	1.8%	2.0%
+90	3.2%	3.2%	3.2%	3.8%	3.3%	3.5%	3.7%	3.5%

[°C]	9	10	11	12	13	14	15	16
−30	0.1%	0.3%	0.1%	0.2%	0.2%	0.8%	1.7%	0.1%
+25	1.7%	1.7%	1.8%	2.1%	1.8%	1.7%	1.6%	1.7%
+90	3.6%	3.6%	3.8%	4.1%	3.3%	3.5%	4.0%	3.7%

very close to 0 for most of the devices), the reason is different. Our observation is that at these temperatures the SRAM start-up patterns exhibit a significant decrease in the Hamming weight for all tested devices. For all devices the Hamming weight was very close to 0, which means that almost all bits of the memory start-up as a 0 and only very few (in the order of magnitude of 1%) as a 1. These results clearly show that by exposing the PIC16F1825 to (extremely) low temperatures it is possible to make the start-up pattern of the SRAM more predictable. We shall call this controlled decrease of entropy a "freezing attack". Such an attack scenario is outside the scope of our attacker model (see Section 4.1), since it requires for an attacker to have physical access to the device to freeze the memory. However, this phenomenon does present a major issue for usability of the PIC16F1825, because it will not be possible to generate sufficient entropy for the PRNG seed when ambient temperatures are sufficiently low (e.g. during wintertime in large parts of the world).

Based on the problems detected at low temperatures, the clearly visible patterns within the SRAM start-up values (see Fig. 3), and the very small security margin hinted

at by the min-entropy calculations, we advise strongly against using PIC16F1825 devices to generate a secure seed for PRNG initialization[6].

3.5 Discussion of measurement results

From the measurement results in the two previous sections it becomes clear that the two investigated device types behave very differently. The STM32F100R8 devices show great results when it comes to deriving entropy from the noise on SRAM start-up patterns, while the PIC16F1825 are clearly unfit for the purpose of extracting a truly random seed from this noise. Besides the simple conclusion that when one wants to implement a PRNG on a microcontroller, which uses a truly random seed derived from SRAM start-up noise, one should not use the PIC16F1825 but rather the STM32F100R8 devices, this section also takes a closer look at trends that become apparent from the results of these two devices.

Dependencies on ambient temperature. Fig. 4 provides a visual representation of the results of the min-entropy measurements from the STM32F100R8 and PIC16F1825 chips at different temperatures from Table 1 and Table 2. In this plot, measurements at the same temperature are encoded using the same shape for data points.

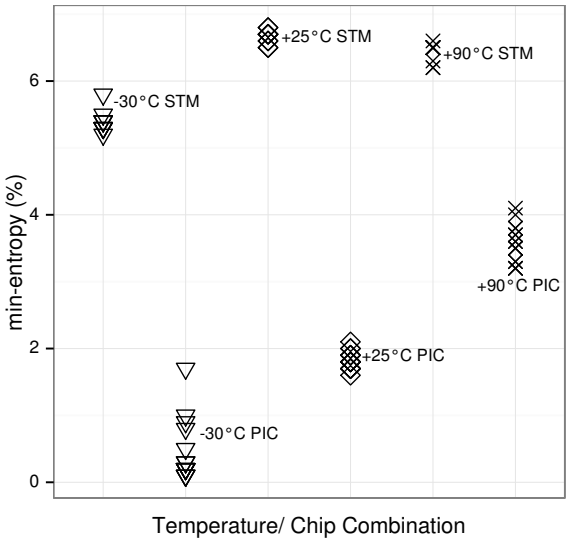

Fig. 4. Scatter plot of min-entropy at various temperatures for STM32F100R8 and PIC16F1825.

[6] A less extensive evaluation of PIC16F877A and PIC16F721 devices seems to suggest that other devices in the PIC16F family have similarly low entropy in their SRAM start-up patterns.

From Fig. 4 we can derive two clear trends. The first is, as already concluded before, the fact that the min-entropy of the STM32F100R8 memories is greater than that of the PIC16F1825s under any tested circumstance. More interesting is the second trend, which shows that the behavior over different temperatures for these two devices is very different. While the min-entropy of the STM32F100R8 is reasonably stable over temperature, the min-entropy of the PIC16F1825 shows a clear (perhaps even linear) correlation with the temperature. In other words: the colder the ambient temperature, the lower the min-entropy of these start-up patterns.

This behaviour again shows that the PIC16F1825 devices are very much unsuitable for use in the proposed random number generator. An uncontrollable variable, such as the ambient temperature, should never be able to influence the amount of entropy that will be available in the seed of a PRNG.

Dependencies on start-up power curve. During our experiments we have made an attempt to increase the min-entropy in the start-up patterns of the PIC16F1825 devices (at room temperature) by making alterations to their power circuitry. Since altering circuitry goes against the principle employed in this paper, i.e. using unmodified COTS devices, we will not consider these results in the main story of the paper. More details on this power-up curve dependency can be found in Appendix A. However, we would like to point out that altering the shape of the power-up curve on the supply pins of the PIC16F1825 devices has resulted in a reduction of the bias in the SRAM start-up patterns, which increases the entropy of these memories. This observation shows the considerable possibility that the biasing in the start-up pattern is caused by internal circuitry that is in charge of supplying power to the SRAM. It is possible that (analog) components inside the PIC16F1825 distort the supply curve before it is able to power-up the SRAM. Unfortunately Microchip does not provide information about their silicon implementation, which makes it impossible for us to verify what is happening inside the devices.

4 Architecture of an SRAM-based RNG

SRAM start-up values, as analyzed in the previous section, can be used to derive PRNG seeds in an efficient and lightweight manner on low-cost COTS microcontrollers without the need for extra hardware (along the lines of [10]). In a nutshell, we measure the start-up contents of SRAM cells right after power up and post-process them in order to extract a seed (see Fig. 5):

1. First, the device is powered up. Care should be taken that the power-up voltage follows a nice curve, as explained in Appendix A. Furthermore, the device should be properly discharged before power-up, such that the SRAM is completely cleared.
2. In the second step, the seed generation algorithm is run: the code reads the entire SRAM content and applies a hash function to it to derive the seed, as suggested in [3, 5, 9]. This step ensures a consistently high entropy in the seed value. Note that this algorithm must be the first code that executes on the device in order to ensure that the SRAM contains uninitialized data.

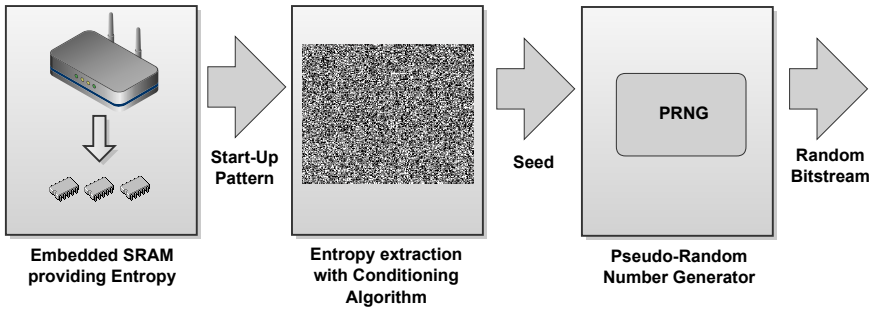

Fig. 5. Secure PRNG seeding based on noise in SRAM start-up pattern.

Care should be taken when implementing the hash function: first of all, the hash function must be designed to be lightweight, as the target embedded platform probably has limited storage and computational power. Furthermore, any temporary storage used by the hash function will overwrite parts of the SRAM start-up patterns, which need to be excluded from seed derivation (in the worst case 8 bits of entropy from the SRAM start-up pattern are lost for each byte used in the hash function implementation). Therefore, it is important to select a hash function that requires a small program size and has limited memory consumption; Appendix B discusses suitable implementations of the hash function.

3. Finally, the generated hash is used to seed a PRNG, which can then be used to obtain a stream of random numbers. Implementations for PRNGs are extensively documented in the literature (for example, see [3, 5, 9]). For use in low-cost devices we suggest to apply a block cipher in counter (CTR) or output feedback (OFB) mode, which are known to before as cryptographically secure PRNGs. The reason for this choice is that a block cipher implementation is most likely already available in a device which requires cryptographic algorithms; this reduces both implementation costs and code size compared to implementing a dedicated PRNG algorithm. With the appropriate construction, a block cipher can also be used as a hash function, further decreasing costs and code size (see Appendix B).

4.1 Security considerations and attacker model

Crucial for security is to maintain the unpredictability of the data stream produced by the PRNG. Once an attacker knows the seed, the entire stream becomes predictable. Thus, care needs to be taken that no other algorithm has access to the seed value — approaches to achieve this are the subject of a separate field of embedded cryptography research and thus outside of the scope of this paper.

In this work we assume an attack scenario in which an adversary has no direct physical access to the microcontroller. Otherwise it would be impossible to ensure that the power-up SRAM value remains secret, since an adversary can use a debugging

interface such as JTAG to halt the microcontroller during start-up, read out the data and then let the start-up process continue.

To limit the exposure of the initial SRAM state and prevent attacks where the seed is re-calculated from SRAM content, all SRAM (except for the seed value) should be cleared immediately after seed generation. This can be achieved by making sure that the seeding algorithm is the very first code that runs on power-up and that the algorithm is executed atomically. Methods to ensure this, such as disabling interrupts and preventing unauthorized firmware modifications, are outside the scope of this paper.

Finally, in order to guarantee proper SRAM resets in between power-cycles of the microcontroller, care should be taken that the microcontroller's positive supply lines are grounded when the device shuts down. If this is not done, it might power up with old, predictable data with low entropy still present in SRAM.

5 Conclusion

In this work, the problem of weak seeds used to initialize PRNGs was addressed. Such weak seeds lead to the PRNG generating predictable random numbers. We presented a lightweight software-only approach to generate secure seeds on commercial off-the-shelf microcontrollers. The source of entropy used to generate these seeds is the noise present in SRAM at start-up. In order to support that such an approach is feasible with COTS microcontrollers, we measured and evaluated this noise at various ambient temperatures in two popular devices, the STMicroelectronics STM32F100R8 (an ARM Cortex-M3) and the Microchip PIC16F1825. Our analysis shows that the SRAM start-up patterns of the PIC16F1825 devices contain very little entropy, which are thus unfit for secure seed generation. Furthermore, we address the peculiarities of these devices under both temperature and supply voltage variations. The SRAM start-up patterns of the STM32F100R8 devices on the other hand contain a large amount of entropy, thereby showing that our approach is indeed feasible and that unmodified COTS microcontrollers using a software-only approach can generate secure seeds for PRNGs.

Acknowledgements

This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007) and by the Flemish iMinds projects. In addition, this work is supported in part by the Hercules Foundation AKUL/11/19, and by the European Commission through the ICT programme under contract FP7-ICT-2011-284833 PUFFIN.

Bibliography

[1] Babaei, M., Farhadi, M.: Introduction to Secure PRNGs. International Journal of Computer Network and Security 4(10), 616–621 (2011)

[2] Balasch, J., Ege, B., Eisenbarth, T., Gérard, B., Gong, Z., Güneysu, T., Heyse, S., Kerckhof, S., Koeune, F., Plos, T., Pöppelmann, T., Regazzoni, F., Standaert, F.X., Assche, G.V., Keer, R.V., van Oldeneel tot Oldenzeel, L., von Maurich, I.: Compact Implementation

and Performance Evaluation of Hash Functions in ATtiny Devices. IACR Cryptology ePrint Archive 2012, 507 (2012)

[3] Barker, E., Kelsey, J.: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. NIST Special Publication 800-90A (Jan 2012)

[4] Dingledine, R.: Tor Security Advisory: Debian Flaw Causes Weak Identity Keys (May 2008), https://lists.torproject.org/pipermail/tor-announce/2008-May/000069.html

[5] Eastlake, D., Schiller, J., Crocker, S.: Randomness Requirements for Security. RFC 4086 (Best Current Practice) (Jun 2005), http://www.ietf.org/rfc/rfc4086.txt

[6] Guajardo, J., Kumar, S.S., Schrijen, G.J., Tuyls, P.: FPGA Intrinsic PUFs and Their Use for IP Protection. In: Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems. Lecture Notes in Computer Science, vol. 4727, pp. 63–80 (Sep 2007)

[7] Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In: Proceedings of the 21st USENIX Security Symposium (Aug 2012)

[8] Holcomb, D.E., Burleson, W.P., Fu, K.: Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. IEEE Trans. Comput. 58(9), 1198–1210 (Sep 2009)

[9] Kelsey, J., Schneier, B., Ferguson, N.: Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. In: Proceedings of the 6th Annual International Workshop on Selected Areas in Cryptography. pp. 13–33. SAC '99 (1999)

[10] van der Leest, V., van der Sluis, E., Schrijen, G.J., Tuyls, P., Handschuh, H.: Efficient Implementation of True Random Number Generator Based on SRAM PUFs. In: Naccache, D. (ed.) Cryptography and Security. Lecture Notes in Computer Science, vol. 6805, pp. 300–318. Springer (2012)

[11] Lenstra, A.K., Hughes, J.P., Augier, M., Bos, J.W., Kleinjung, T., Wachter, C.: Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064 (2012)

[12] Maes, R.: Physically Unclonable Functions: Constructions, Properties and Applications. Ph.D. thesis, KU Leuven (2012), Ingrid Verbauwhede (promotor)

[13] ARM Holdings PLC: Results for the Fourth Quarter and Full Year 2012 (Feb 2013), http://ir.arm.com

[14] Databeans Inc.: Microcontroller Market Share: In 3 Dimensions (Mar 2012)

[15] Debian Security: DSA-1571-1 OpenSSL – Predictable Random Number Generator. Tech. rep. (May 2008), http://www.debian.org/security/2008/dsa-1571.en.html

[16] Information Technology Laboratory NIST: FIPS 140-3: Security Requirements for Cryptographic Modules, Annex A: Approved Security Functions for FIPS PUB 140-3 (Jul 2009), draft

[17] Moro, T., Saitoh, Y., Hori, J., Kiryu, T.: Generation of Physical Random Number Using the Lowest Bit of an A-D Converter. Electronics and Communications in Japan (Part III: Fundamental Electronic Science) 89(6), 13–21 (2006)

[18] Mowery, K., Wei, M., Kohlbrenner, D., Shacham, H., Swanson, S.: Welcome to the Entropics: Boot-Time Entropy in Embedded Devices. In: IEEE Symposium on Security and Privacy (May 2013), to be published

[19] Petrie, C., Connelly, J.: A Noise-based IC Random Number Generator for Applications in Cryptography. Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on 47(5), 615 –621 (May 2000)

A Dependencies of SRAM Entropy on Start-up Power Curve

In Section 3.4, we have seen that the SRAM start-up patterns of the tested PIC16F1825 devices showed severe biasing at room temperature. Based on the authors' practical experience, the most common reason for biased start-up patterns (besides those due to low temperatures, which were already shown in the same section) is a due to the suppy voltage ramp-up curve on the SRAM cells. Measuring the supply voltage curve applied to the PIC16F1825 by our controller board shows a ramp as can be seen on the left side of Fig. 6. Unfortunately, this is a ramp-up curve which is very common for powering SRAMs and usually results in proper (unbiased) start-up patterns. However, this does not mean that the supply voltage curve on the actual SRAMs cannot be the root-cause. We can only measure the curve on the external pins of the devices and we do not know what happens internally with the supply voltage before it reaches the SRAM. It is possible that there are (analog) components connected to the power supply, which distort the ramp-up on the SRAM. Unfortunately Microchip does not provide information about their silicon implementation, which makes it impossible for us to verify what is happening inside the devices.

In an attempt to make the start-up pattern of the SRAMs more random, we have performed experiments with varying the shape of the supply voltage curve on the supply pins. One of the shapes that we have tried (and the only one that improved our results) consists of a short pulse on the supply pin just before the actual ramp-up curve. An example of such a shape can be seen on the right side of Fig. 6. In this shape the height and width of the pulse can be varied.

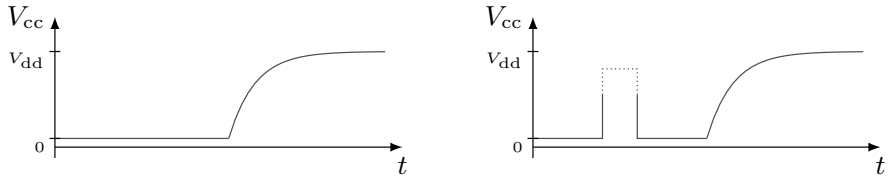

Fig. 6. Original supply voltage curve (*left*, normal for SRAMs) and altered curve (*right*).

With the "new" supply voltage curve on the supply pins of the PIC16F1825s, some devices presented start-up patterns which turned out to be more random than the original patterns. An example of such a new pattern can be found in Fig. 7. Unfortunately, we were not able to make the start-up patterns of all PIC16F1825 devices more random with this method. Also, the height of the pulse before the voltage ramp-up curve (which resulted in more random patterns for some devices) was different for each individual device.

The experiments described in this appendix are still in a very preliminary stage and will be part of future work in the ongoing studies on this topic. What they do show however, is that the supply voltage curve on an SRAM can have a big impact on the behaviour of its start-up pattern. Therefore, it is be very important when implementing

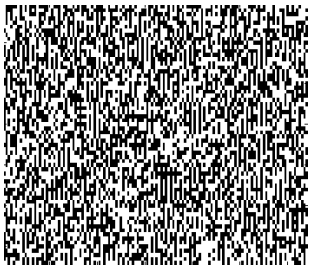

Fig. 7. Example start-up pattern of PIC16F1825 (1 KiB SRAM) with altered supply voltage power-up curve at +25 °C. White represents a bit with value 0, black a bit with value 1.

a secure PRNG seed generator as described in this paper to make sure that the supply voltage curve to the COTS microcontroller allows for good random noise behaviour in the SRAM start-up pattern.

Furthermore, based on these experiments we can think of at least one additional attack scenario: a hardware system that supplies a modified power-up curve to the microcontroller. It is possible on certain devices, with a specific power-up curve, to reduce the entropy contents of the SRAM to a minimum. Such an attack could be imagined on a cellphone, in which end-users can easily replace the battery, thus we call this the "evil battery" attack. Possible countermeasures are the use of power management and power monitoring circuits.

B On the Selection of a Hash Function

In this section, we discuss the selection of a hash function implementation for secure seed generation. We make a suggestion for what is, in our opinion, the best choice, based on memory and code space efficiencies.

Any examples we give, we only give for the STM32F100R8. Due to the extremely low entropy in its SRAM start-up values (0.1% at −30 °C), the PIC16F1825 is not usable for secure PRNG generation using the suggested approach. Due to entropy being lost to temporary storage of the hash function, any conceivable hash function implementation will reduced the available entropy in the PIC16F1825 to 0.

Since it would fall outside the scope of this work, we did not implement any hash functions. Instead, we base our recommendations on Balasch et al. [2], in which a wide selection of 25 different hash functions, written in hand-optimized assembly, are presented. The implementations in [2] are for an 8-bit Atmel AVR chip, and thus not directly transferable to the chips that we have investigated. However, the required code sizes can be used as a relative size indicator. More important than code size though is the required amount of SRAM for temporary storage, which luckily is independent of the hardware architecture.

The required amount of SRAM for the hash function influences the total remaining entropy in the SRAM start-up pattern. No assumptions are made about the distribution of entropy within the SRAM, and therefore one has to assume a worst case scenario in

which every byte of storage used by the hash function removes a full 8 bits of entropy from the total entropy available in the SRAM start-up pattern.

Assuming a required hash digest size of 256 bit, as required for FIPS 140-3 [16] compliance, the hash function, of those presented in [2], that requires the least storage is S-Quark, with 69 bytes. An alternative is PHOTON-256/32/32, which requires 82 bytes. If one prefers to use the newest SHA3 algorithm, then Keccak with parameters $r = 144, c = 256$ is the choice that requires the least amount of storage (114 bytes). Instead of using a dedicated hash function, it is also possible to use a block cipher-based hash construction. In that case, a Hirose/AES-256 construction is ideal, since it requires only 104 bytes of storage. These requirements, together with the remaining min-entropy in the STM32F100R8, are listed in Table 3.

Table 3. Hash function SRAM requirements and influence on SRAM entropy in STM32F100R8 (8 KiB SRAM). SRAM consumption data from [2]. Remaining min-entropy based on a pessimistic min-entropy estimate of 3% (see Section 3.3).

Hash	**SRAM [byte]**	**Remaining min-entropy [bit]**
S-Quark	69	1 414
PHOTON-256/32/32	82	1 310
Keccak$[r = 144, c = 256]$	114	1 054
Hirose/AES-256	104	1 134

It is obvious from Table 3 that the STM32F100R8 microcontroller has plenty entropy left over in its SRAM start-up pattern for any of the chosen hash function implementations. The formula used to calculate the remaining amount of entropy in SRAM for a given amount of memory consumption can be inverted to calculate the maximum allowed amount of SRAM consumption when requiring a minimum remaining entropy of 256 bit:

$$\begin{aligned} & 256 \leq 8 \cdot \ (8192 \cdot 0.03 - x) \\ \Leftrightarrow\ & x \leq 8192 \cdot 0.03 - \frac{256}{8} \\ \Leftrightarrow\ & x \leq 213.76, \end{aligned} \tag{1}$$

i.e. a hash function implementation on the STM32F100R8 can use a maximum of 213 bytes of SRAM when it is required that at least 256 bits of entropy remain under worst case conditions[7]. Thus, for the STM32F100R8, the choice of hash function will probably not need to depend on the used amount of SRAM, since enough entropy is likely available. For this microcontroller, the algorithm characteristics to look at then would be either required code side or execution time, depending on the application.

Note that, as mentioned in Section 4, block ciphers in CTR or OFB mode can be used as PRNGs. Thus, considering the benefits of code size reduction and a smaller

[7] Worst case conditions assume that every byte of SRAM used by the hash function reduces the total available entropy by 8 bits and that there is only 3% entropy in the SRAM start-up pattern of the STM32F100R8.

codebase to debug, we recommend the use of a block cipher both for hashing and as an PRNG. An obvious choice for a block cipher to use is AES, due to the facts that it is an internationally accepted standard, has been thoroughly studied and not been found vulnerable to attacks, and many optimized implementations exists for a wide range of platforms.