Efficient Implementation of True Random Number Generator based on SRAM PUFs

Vincent van der Leest, Erik van der Sluis, Geert-Jan Schrijen, Pim Tuyls, and Helena Handschuh

Intrinsic-ID, Eindhoven, The Netherlands
http://www.intrinsic-id.com

Abstract. An important building block for many cryptographic systems is a random number generator. Random numbers are required in these systems, because they are unpredictable for potential attackers. These random numbers can either be generated by a truly random physical source (that is non-deterministic) or using a deterministic algorithm. In practical applications where relatively large amounts of random bits are needed, it is also possible to combine both of these generator types. A non-deterministic random number generator is used to provide a truly random seed, which is used as input for a deterministic algorithm that generates a larger amount of (pseudo-)random bits. In cryptographic systems where Physical Unclonable Functions (PUFs) are used for authentication or secure key storage, an interesting source of randomness is readily available. Therefore, we propose the construction of a FIPS 140-3 compliant random bit generator based on an SRAM PUF in this paper. These PUFs are a source of instant randomness, which is available when powering an IC. Based on large sets of measurements, we derive the min-entropy of noise on the start-up patterns of SRAM memories. The min-entropy determines the compression factor of a conditioning algorithm, which is used to extract a truly random (256 bits) seed from the memory. Using several randomness tests we prove that the conditioned seed has all the properties of a truly random string with full entropy. This truly random seed can be derived in a low cost and area efficient manner from the standard IC component SRAM. Furthermore, an efficient implementation of a deterministic algorithm for generating (pseudo-)random output bits will be proposed. Combining these two functions leads to an ideal way to generate large amounts of random data based on non-deterministic randomness.

1 Introduction

Physical Unclonable Functions (PUFs) are most commonly used for identification or authentication of Integrated Circuits (ICs), either by using its inherent unique device fingerprint or by using it to derive a device unique cryptographic key. Due to deep-submicron manufacturing process variations every transistor in an IC has slightly different physical properties that lead to measurable differences in terms of its electronic properties e.g. threshold voltage, gain factor, etc. Since these process variations are uncontrollable during manufacturing, the physical properties of a device can neither be copied nor cloned. It is very hard, expensive and economically not viable to purposely create a device with a given electronic fingerprint. Therefore, one can use this inherent physical fingerprint to uniquely identify an IC.

However, there is another possible application area for PUFs. This application is the generation of random numbers, which can be done based on the noise properties of PUFs. As studies on identification of ICs using PUFs have proven, measurements of the physical structures that make up PUFs are always noisy. This means that two measurements of the same PUF (under the same external conditions) will always result in two slightly different outputs. In case of SRAM PUFs, where this paper focusses on, two measurements of start-up patterns on an SRAM will result in two patterns that are very similar. However, several bits have a different value between the two measurements. In order to be able to use this phenomenon for random number generation, a sufficient amount of bits should change between measurements. Furthermore, the bits that have changed should be unpredictable. In this publication we will prove that both of these requirements are met for the SRAM memories that we have measured. Therefore, we will propose a construction for utilizing this random behaviour of SRAM PUFs in a streaming random number generator that is FIPS 140-3 compliant.

1.1 Related Work

In 2001, Pappu [15] introduced the concept of PUFs under the name Physical One-Way Functions. The indicated technology is based on the response (scattering) obtained when shining a laser on a bubble-filled transparent epoxy wafer. Gassend et al. introduce Silicon Physical Random Functions [4] which use manufacturing process variations in ICs with identical masks to uniquely characterize each IC. The statistical delay variations of transistors and wires in the IC were used to create a parameterized self oscillating circuit to measure frequency which characterizes each IC. This circuit is nowadays known as a Ring Oscillator PUF. Another PUF based on delay measurements is the Arbiter PUF, which was first described by Lee et al. in 2004 [9]. Besides hardware intrinsic PUFs based on delay measurements a second type, based on the measurement of start-up values of memory cells, is known. This type includes SRAM PUFs introduced by Guajardo et al. in 2007 [5], so-called Butterfly PUFs introduced in 2008 by Kumar et al. [8] and finally D flip-flop PUFs also introduced in 2008 by Maes et al. [10]. Implementations of these PUF types exist for dedicated ICs, programmable logic devices such as Field Programmable Gate Arrays (FPGAs) and also for programmable ICs such as microcontrollers.

Random number generation based on non-deterministic physical randomness in ICs has been the topic of several publications. Examples of physical sources that have been studied for randomness extraction are amplification of thermal noise on resistors [16], sampling of free running oscillators [16], and noise on the lowest bits of AD converters [12].

Finally, several papers have been written about using PUFs for random number generation. These papers focus on deriving randomness from meta-stable behaviour in PUFs that are either based on measurements of delay lines [14, 11] or on the start-up patterns of memory cell [6, 7]. The work in this paper is also based on this latter form of physical randomness.

1.2 Our Contribution

In this paper we propose the construction of a streaming random number generator based on SRAM PUFs that is compliant to the FIPS 140-3 standard as specified by the National Institute of Standards and Technology (NIST). This construction is based on a truly random seed derived from noise on the start-up pattern of SRAM memory, which is used as input for a deterministic random bit generator (DRBG). Sections 2 and 5 contain more details on our construction.

Furthermore, we take another look at the calculation (as has been proposed in [7]) of the min-entropy that can be extracted from noise on SRAM start-up patterns. Based on our findings and specifications from the NIST Special Publication 800-90 [1] we recommend a different approach for calculating this min-entropy.

Finally, this paper continues the statistical testing of the extracted random bits that was started in [7]. This is done by expanding the set of randomness tests that are performed on the data derived by conditioning. The test results that can be found in section 4 of this paper will provide the reader with confidence that the seed supplied to the DRBG is generated by a truly random source with an entropy that is at least as high as the length of the seed.

1.3 Paper Organisation

After this introduction we proceed to section 2 that contains a description of the random number generator construction that is the main focus of this publication. In section 3 can be found how we have determined the min-entropy of several SRAM memory types. This derived min-entropy is used to design a conditioning algorithm that provides the deterministic part of the random number generator with a truly random seed. This algorithm is described in section 4 and uses the min-entropy as a lower bound for the amount of randomness that can be extracted from the noise on SRAM start-up patterns. After performing conditioning on a large set of measurements, randomness tests are performed on the output of this algorithm. The results from these tests can also be found in section 4. Section 5 contains the specifications of our hardware implementation of this random number generator can be found. Finally, the conclusions of this publication are gathered in section 6.

2 PUF based random number generation

Many cryptographic protocols depend on the availability of good random numbers. Random numbers are important for such protocols because they cannot be predicted by potential attackers. For example in key generation algorithms or key agreement protocols a privately generated random number should not be predictable by other parties, since this would be a serious threat to the security of the system.

Random numbers for cryptographic applications can be generated using two different basic approaches. The first approach is to derive bits from a truly random physical source that is non-deterministic. The output bits of such a source are random because of some underlying physical process that has unpredictable behaviour. Examples of such true random sources in ICs are for example free running oscillators connected to a shift register [16], noise on the lowest bits of AD converters [12], etc. A second approach is to compute random bits using a deterministic algorithm. In this case a sequence of pseudo-random bits is generated from a seed value. Such generators are known as deterministic random bit generators (DRBGs). For an observer who does not have any knowledge about the used seed value, the produced output bits are unpredictable. For a given seed value the produced output bits are completely predictable and hence not random. Therefore the seed value needs to be chosen randomly and needs to be kept secret.

In practical applications where relatively large amounts of random bits are needed, it makes sense to combine both types of random number generators. A true random number generator is used to provide a good random seed. Subsequently this seed value is fed into a DRBG that generates a larger amount of (pseudo-)random bits from this seed. This makes sense since typically the derivation of a true random number is much slower than generating a pseudo-random sequence. The output of most non-deterministic random processes needs post-processing before it is truly random. A DRBG can generate streams of random bits very quickly once a truly random seed is available. Therefore a DRBG is much more efficient for generating large amounts of random bits.

In cryptographic systems where Physical Unclonable Functions (PUFs) are used for authentication or secure key storage, an interesting source of randomness is readily available. Besides using the device-unique characteristics of PUFs, their inherent noisiness can also be used for security purposes. This was already noted by Holcomb et al. in [6, 7].

In this paper we propose the construction of a FIPS 140-3 compliant random bit generator based on an SRAM PUF. Because of the design of an SRAM memory, a large part of the bit cells is skewed due to process variations and tends to start up with a certain preferred value. This results in a device-unique pattern that is present in the SRAM memory each time it is powered. Bit cells that are more symmetrical in terms of their internal electrical properties on the other hand tend to sometimes start up with a '1' value and sometimes with a '0' value. Hence, these bit cells show noisy behaviour. This noise can be used for the generation of random bit streams.

The proposed random number generator construction is depicted in Fig. 1 and consists of two main parts:

1. An SRAM memory connected to a conditioning algorithm for deriving a truly random seed.
2. A deterministic random bit generator (DRBG) according to the NIST 800-90 [1] specification.

The conditioning algorithm is used to derive a truly random seed from the SRAM start-up values. As explained above, only part of the SRAM bits have noisy behaviour when being powered. The entropy in these noisy bits needs to be condensed into a full entropy random seed. The conditioning algorithm takes care of this. Basically, the conditioning algorithm is a compression function that compresses a certain amount of input data into a smaller fixed size bit string. The amount of compression required for generating a full entropy true random output string is determined by the min-entropy of the input bits (see section 3).

The deterministic random bit generator is built according to the NIST 800-90 specification [1]. Internally it uses a so called *instantiate* function and a *generate* function. The instantiate function takes the truly random seed input and generates a first internal state value. Upon request for a certain amount of random output bytes, the generate function uses a deterministic algorithm that derives these bytes from the current state value and updates the internal state value afterwards. For

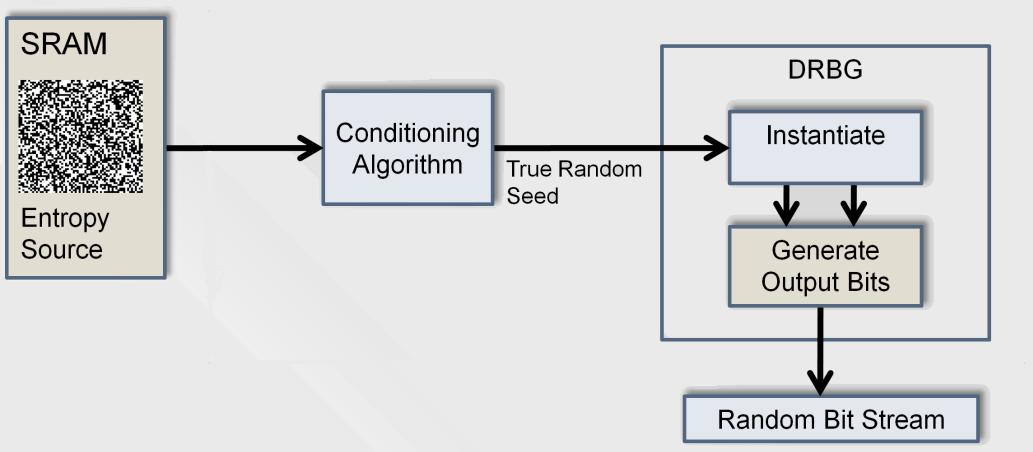

Fig. 1. Construction of SRAM PUF based random number generator.

example, a cryptographic hash function or a block cipher can be used for generating the output bits from the internal state value [1]. The number of output bits that can be generated before requiring reseeding of the DRBG (i.e. by repowering the SRAM and repeating the conditioning and instantiation) depends on the size of the input seed value.

In this paper we focus on the first part of the construction: the generation of the truly random seed from SRAM start-up values. For conditioning we propose a construction based on a cryptographic hash function (see section 4). First, section 3 analyses how much input data is required by estimating the min-entropy of the noise in start-up patterns from different SRAM memories.

3 Min-entropy

As shown in the previous section the noise present in SRAM start-up patterns will be used to derive a seed for a pseudo-random number generator. To make sure that this seed is truly random, the SRAM pattern should be conditioned (e.g., hashed, compressed, etc.). This conditioning will be discussed in the next section. The first step is to determine how much entropy (a mathematical measure of disorder, randomness or variability) is present in the noise of SRAM start-up patterns. In order to derive a minimum for the amount for this randomness, we will be using min-entropy. According to NIST specification [1] the definition of min-entropy is the worst-case (i.e., the greatest lower bound) measure of uncertainty for a random variable.

3.1 Method of deriving min-entropy

For a binary source, the possible output values are 0 and 1. Both of these values have a probability of occurring p_0 and p_1 respectively (the sum of these two probabilities is 1). When p_{max} is the maximum value of these two probabilities, the definition for min-entropy of a binary source is:

$$H_{min} = -log_2(p_{max}) \tag{1}$$

Assuming that all bits from the SRAM start-up pattern are independent, each bit of the pattern can be viewed as an individual binary source. For n independent sources (in this case n is the length of the start-up pattern), the following definition holds:

$$(H_{min})_{total} = \sum_{i=1}^{n} -log_2(p_{i\ max}) \tag{2}$$

Hence, under the assumption that all bits are independent we can sum the entropy of each individual bit. This method is according to NIST specification [1].[1]

3.2 Min-entropy results

To be able to calculate the min-entropy of the noise on different SRAM memories under different environmental conditions, measurements are performed on three different device types. For each type several devices are tested and two of these types contain multiple memory instantiations. In this section we present the results from our measurements.

Since it is known that SRAM start-up patterns are susceptible to external influences (such as varying ambient temperature, voltage variations, etc.), it is important to measure the min-entropy under different circumstances. In this paper we choose to perform measurements at varying ambient temperatures, since this is known to have a significant influence on the start-up patterns [5, 17]. For min-entropy calculations however, the measurement environment for worst-case behaviour should be as stable as possible. Therefore, we will be determining the minimal amount of entropy for each of the individual ambient temperature conditions.

The first device type that is examined is the "PUF-IC", an ASIC that has been designed by Intrinsic-ID and produced through the Europractice program of IMEC. These ICs were produced on a Multi-Project-Wafer (MPW) in 130nm UMC technology and contain three SRAM instantiations that are studied here. Memory 1 is a Faraday SHGD130 memory, memory 2 is Virage asdsrsnfs1p and memory 3 is Virage asssrsnfs1p.

To be able to determine how many measurements (under stable conditions) are required to obtain a good estimate of the min-entropy, 500 measurements are performed on all devices and memories of the "PUF-IC". Two methods are used to calculate the min-entropy for all of these memories at three different temperatures (-40, +20 and +80°C), formula 2 and the method described in [7]. Figure 2 shows the development of the min-entropy value over the number of measurements (for memory 2 of device 1 at +20°C). The shape of the lines in this figure is comparable for all memories, devices and temperatures.

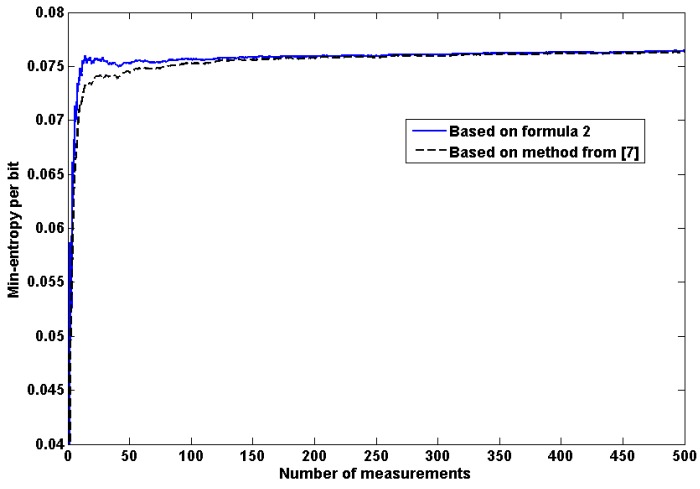

Fig. 2. Two methods for calculating min-entropy of "PUF-IC" memory 2 (device 1, 500 measurements).

[1] In comparison: The method from [7] assumes that each byte of the SRAM is an independent source. It is the author's opinion that there is no reason to assume that there is any correlation between bits from a specific byte. If these bits are uncorrelated we can indeed use the method as described above.

From this figure we draw the following conclusions:

- The min-entropy of noise on SRAM start-up patterns can be calculated using approximately 100 measurements. This conclusion is based on the asymptotic behaviour of the lines in the figure. Therefore, other devices can be evaluated if they have at least 100 measurements per memory (per temperature).
- The resulting min-entropy calculated using formula 2 and the method from [7] are approximately equal. The only difference (for some memories) is that our method converges slightly quicker to the asymptotic value of the min-entropy. Therefore, we will continue using formula 2 to derive min-entropy.

The results of all calculations using formula 2 can be found in table 1. All values in this table are fractional representations of the min-entropy. This means that the amount of min-entropy has been divided by the length of the start-up pattern, resulting in a percentage for the min-entropy.

Table 1. Min-entropy results for PUF-IC memories at different temperatures (based on 500 measurements per device, per memory, per temperature).

	-40^oC			$+20^oC$			$+80^oC$		
Device	Memory 1	Memory 2	Memory 3	Memory 1	Memory 2	Memory 3	Memory 1	Memory 2	Memory 3
1	4.1%	6.1%	6.2%	5.1%	7.0%	6.9%	5.4%	7.6%	7.6%
2	4.7%	5.9%	5.8%	5.2%	5.6%	5.7%	4.1%	2.9%	4.1%
3	4.9%	5.7%	5.8%	5.1%	5.7%	6.5%	4.2%	3.8%	5.3%
4	5.6%	6.5%	6.3%	6.3%	7.3%	6.6%	6.8%	7.9%	6.9%
5	5.2%	5.8%	6.2%	5.8%	6.4%	6.8%	6.5%	7.3%	7.4%

Now that we know that 100 measurements suffice to get an estimate on the min-entropy of the noise on start-up patterns, two other devices are evaluated. The first one is a 65nm device by Cypress with three memories that are all of the type CY7C15632KV18. The results from this device can be found in table 2. The other device that is studied is a 150nm device by Cypress that contains one CY7C1041CV33-20ZSX memory and these results are collected in table 3.

Table 2. Min-entropy results for Cypress 65nm memories at different temperatures (based on 100 measurements per device, per memory, per temperature).

	-40^oC			$+20^oC$			$+80^oC$		
Device	Memory 1	Memory 2	Memory 3	Memory 1	Memory 2	Memory 3	Memory 1	Memory 2	Memory 3
1	4.2%	4.3%	4.4%	4.7%	4.6%	4.7%	5.3%	5.2%	5.3%
2	4.3%	4.3%	4.3%	4.6%	4.6%	4.4%	5.4%	5.4%	5.4%
3	4.0%	4.1%	4.0%	4.7%	4.6%	4.5%	5.3%	5.4%	5.2%
4	4.6%	4.6%	4.6%	4.8%	4.7%	4.8%	5.3%	5.2%	5.2%
5	4.2%	4.2%	3.9%	4.8%	4.8%	4.5%	5.0%	5.6%	5.3%
6	4.3%	4.5%	4.4%	4.5%	4.5%	4.5%	5.1%	5.2%	5.3%
7	4.3%	4.4%	4.4%	4.6%	4.7%	4.6%	5.4%	5.4%	5.2%
8	4.3%	4.3%	4.3%	4.6%	4.7%	4.8%	5.4%	5.4%	5.3%
9	4.2%	4.2%	4.3%	4.7%	4.3%	4.5%	5.3%	5.3%	5.2%
10	4.7%	4.6%	4.4%	4.9%	4.9%	4.5%	5.4%	5.7%	5.3%

3.3 Conclusions from test results

From the results in the previous subsection it becomes clear that all studied memories have different amounts of randomness that can be extracted from noise. In general we can say that the entropy

Table 3. Min-entropy results for Cypress 150nm memories at different temperatures (based on 100 measurements per device, per memory, per temperature).

	-40^oC	$+20^oC$	$+80^oC$
Device	Memory 1	Memory 1	Memory 1
1	2.6%	4.1%	4.2%
2	2.5%	4.1%	4.4%
3	2.5%	4.2%	4.4%
4	2.6%	4.2%	4.4%
5	3.2%	4.2%	4.2%
6	2.9%	4.1%	4.3%
7	2.5%	4.1%	4.3%
8	2.7%	4.1%	4.4%

of the noise is minimal at the lowest measured temperature of -40^oC. Furthermore, min-entropy values from different memories of the same type, measured under comparable external conditions, can either be quite stable (Cypress memories) or varying ("PUF-IC" memories).

Based on the results, we can conclude that for each of the evaluated memories it should be possible to derive a truly random seed (with full entropy) of 256 bits from a limited amount of SRAM. For instance, if we assume a conditioning algorithm that compresses its input to 2% (each studied memory has a min-entropy that is higher than 2% under all circumstances) the required length of the SRAM start-up pattern is 1600 bytes.

4 Truly random seed

As described in section 2, a seed is required prior to generating (pseudo-)random output bits with a DRBG. This seed is used to instantiate the DRBG and determine its initial internal state. According to the specifications by NIST [1], the seed should have entropy that is equal to or greater than the security strength of the DRBG. In our design (as can be found in 5) the DRBG requires an input seed of length 256 bits with full entropy. In order to achieve this a conditioning algorithm can be used. This section describes our chosen algorithm as well as tests that will be performed on the output of the algorithm in order to indicate that the resulting seed has the properties of a truly random 256 bits string (true randomness can never be proven, it is only possible to get an indication whether the source might be truly random).

4.1 Conditioning

To extract a truly random seed (with full entropy) from SRAM start-up noise based on the min-entropy calculations from the previous section, we have selected the SHA-256 hash function [20] to perform conditioning. This hash function compresses input strings with lengths that are a multiple of 512 bits into an output string of length 256 bits. In order for this output to have full entropy, the amount of entropy at the input of the hash function should be at least 256 bits. For example, if the input string has a min-entropy of 2%, the length of this input needs to be at least 1600 bytes (= 25 blocks of 512 bits).

Selecting the SHA-256 hash function for conditioning is based on the following reasons:

- The output of SHA-256 is 256 bits, which is equal to the required input length for our DRBG algorithm (under the constraint that these 256 bits should have full entropy).
- The SHA-256 hash function has been approved by NIST (see [21]).
- The properties of SHA-256 allow us to re-use this hash function in our DRBG algorithm. This is important for the required area in our hardware implementation (see section 5).

4.2 Tests performed on seed

This subsection describes the tests that are performed on the data generated from SRAM start-up patterns by the conditioning algorithm to indicate randomness. In order to have a significant set of data for the randomness tests 500.000 measurements have been performed on Memory 2 of "PUF-IC" device 1 at $+20^o$C. These measurements are conditioned using the SHA-256 hash function, with 640 bytes of input data per 256 bits output, resulting in 128.000.000 conditioned bits. Since this memory has a min-entropy of 7% at $+20^o$C, we use the conditioning algorithm to compress to 5% (this is slightly below the value of the min-entropy). All tests described below are performed on this data set.

Hamming Distance test The first test that will be performed is the Hamming Distance test. This test is used to get a first impression whether the seeds derived by the conditioning algorithm might be truly random. For this test a subset of the generated seeds (in this case 5000) are compared to each other based on fractional Hamming Distance (HD), which is defined by:

$$\mathrm{HD}(\bar{x}, \bar{y}) = \sum_{i=1}^{n} x_i \oplus y_i \tag{3}$$

$$\text{Fractional HD}(\bar{x}, \bar{y}) = \frac{\sum_{i=1}^{n} x_i \oplus y_i}{n} \tag{4}$$

Based on formula 4, where $\bar{x}$ and $\bar{y}$ are SRAM start-up patterns with their individual bits x_i and y_i respectively, a set of fractional HDs is composed. Comparing all 5000 measurements to each other results in 12.497.500 HDs. To indicate that these seeds have been created by a random source, the set HDs should have a Gaussian distribution with mean value 0.5 and a small standard deviation.

CTW test The second test that will be performed on the resulting seeds is a compression test. The Context-Tree Weighting method (CTW) [18, 19] is an optimal compression method for stationary ergodic sources. The compression that CTW achieves on bit strings is therefore often used as an estimator for the entropy rate (see, for example, [3]). We use the CTW compression method as follows. If the CTW algorithm manages to (losslessly) compress a set of 50.000 seeds[2] that has been created by the conditioning algorithm, this indicates that the seeds do not have full entropy. Hence, assuming that the seeds are created by a true random source with at least 256 bits of entropy, the CTW algorithm should not be able to compress the data.

NIST tests As a final step to evaluate the randomness of the generated seeds, the complete set of randomness tests from the Special Publication 800-22 issued by NIST [13] will be used. These tests can be considered to be the current state of the art with regard to randomness testing. In the tests we evaluate the passing ratio of a varying number of input strings and string lengths. When the number of inputs is n and the probability of passing each test is p , then the number of strings that pass the test follows a binomial distribution. The significance level of each test in NIST SP 800-22 is set to 1%, which means that 99% of the test samples should pass the tests ($p = 0.99$) if the samples are generated by a truly random source. For example, when the number of strings is 125 the value of p' (observed ratio to pass test) should be between the following values:

$$p' \geq p - 3\sqrt{\frac{p(1-p)}{n}} = 0.99 - 3\sqrt{\frac{0.99 \times 0.01}{125}} = 0.9633 \tag{5}$$

Furthermore, a P-value is introduced to evaluate whether the sequence of results per randomness tests are uniformly distributed in order to indicate randomness. This uniformity is determined by a χ^2 test, which produces the P-value. In order to indicate randomness, this P-value should be at least 0.0001 [13]. Therefore a NIST test with 125 input samples is only passed when:

$$p' \geq 0.9633 \ \cap \ \text{P-value} \geq 0.0001 \tag{6}$$

[2] This is the maximum number of input bits allowed for our (Matlab) implementation of CTW.

4.3 Test results

Hamming Distance test Figure 3 shows the Hamming Distance distribution of the 5000 seeds generated by the conditioning algorithm. As can be seen in this figure, the distribution is perfectly Gaussian with a mean value μ of 0.5. The standard deviation σ of this distribution is 0.03125. Using the following formula from [2], an estimation of the entropy of this data set can be made:

$$\text{Entropy} = \frac{\mu * (1 - \mu)}{\sigma^2} = \frac{(0.5)^2}{(0.03125)^2} = 256 \text{ bits} \tag{7}$$

Based on this evaluation of 5000 seeds it appears that the output strings of the conditioning algorithm are truly random and contain full entropy, since the length of these strings is 256 bits.

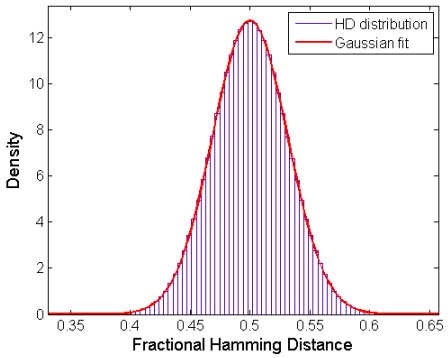

Fig. 3. Hamming Distance distribution over 5000 generated seeds.

CTW test The result of the CTW test can be found in table 4. From this result it becomes clear that the CTW algorithm is unable to compress the seeds that are generated by the conditioning algorithm. Therefore, this is another indication that the generated seeds could be truly random and there is no reason to assume that they do not contain full entropy.

Table 4. CTW compression results on data from conditioning algorithm

Input data	Input length	Minimal output length	Compression ratio
Concatenation of 50.000 seeds	12.800.000	12.800.011	$\geq$ 100%

NIST tests The results from the performed NIST randomness tests are shown in table 5. In this table can be seen that the strings generated by the conditioning algorithm pass all the NIST randomness tests that we are able to perform. Note that we have limited our test set to those tests that can be performed with the amount of test data that is available. As stated in the introduction of this section, the total number of conditioned bits available for testing is 128.00.000. In order to create three different data sets for NIST testing, these bits have been divided in strings of different lengths. The first data set contains 125 strings of length 1.024.000, because the strings need to be at least 1.000.000 bits long to be able to perform all tests from the NIST suite. To be able to run the NIST tests on a larger number of strings, two data sets are created with more strings (of shorter length). These sets contain 250 strings of 512.000 and 1000 strings of 128.000 bits respectively. The NIST randomness tests that cannot be performed, due to insufficient string lengths, are omitted (and are denoted by n.a. in the table).

Table 5. Test results of NIST randomness test suite.

Test	Minimal required input size per string	Settings	Set 1 125 strings 1024000 bits	Set 2 250 strings 512000 bits	Set 3 1000 strings 128000 bits
Frequency	100		PASS	PASS	PASS
Cumulative Sum	100		PASS	PASS	PASS
Runs	100		PASS	PASS	PASS
FFT	1000		PASS	PASS	PASS
Longest Run	6272		PASS	PASS	PASS
Block Frequency	12800	M=128	PASS	PASS	PASS
Approximate Entropy	32768	m=10	PASS	PASS	PASS
Rank	38912		PASS	PASS	PASS
Serial	262144	m=16	PASS	PASS	n.a.
Universal	387840		PASS	PASS	n.a.
Random Excursions	1000000		PASS	n.a.	n.a.
Random Exc. Variant	1000000		PASS	n.a.	n.a.
Linear Complexity	1000000	M=500	PASS	n.a.	n.a.
Overlapping Template	1000000	m=9	PASS	n.a.	n.a.
Non-overlap. Temp.	1000000	m=10	PASS	n.a.	n.a.

5 Implementation of the random number generator

Based on the test results, we have created a hardware implementation of the random number generated as proposed in this paper. This random number generator consists of SRAM, the conditioning algorithm and a DRBG that is compliant to FIPS 140-3 as specified in [1]. These building blocks are combined in the implementation as depicted in figure 1.

The amount of SRAM used for the implementation is 2kB, which is based on the min-entropy estimations from section 3. The minimal amount of observed min-entropy in this section is 2.5%. In case an SRAM of 2kB contains 2.5% entropy, it is still possible to extract 409 random bits from this memory. Therefore, a memory of this size should be sufficient for our conditioning algorithm to generate the truly random seed of 256 bits as required by the DRBG (under the circumstances that have been measured for this paper).

As stated in section 4, the conditioning algorithm used in this implementation is the SHA-256 hash function. Our implementation of SHA-256 can perform a hash in 64 clock cycles at a clock frequency of 200MHz. Based on this information, we can calculate the amount of time required to derive the seed by conditioning:

$$\text{Time} = \frac{\text{Number of SRAM bits}}{\text{Input bits/hash}} * \frac{\text{Clocks/hash}}{\text{Clock frequency}} = \frac{2048 * 8}{512} * \frac{64}{200\text{MHz}} = 1.024 * 10^{-5} \text{ sec.}$$

Hence, the amount of time required to derive a random seed from an SRAM PUF is very limited. In other words, the random seed derived from non-deterministic physical randomness can be used (almost) instantly.

This random seed is used as input for the DRBG, which converts the seed into a stream of random bits. The chosen DRBG implementation for our construction is the Hash_DRBG (as described in [1]) based on the SHA-256 hash function. This is because for this DRBG the SHA-256 function can be re-used, which saves additional hardware and therefore minimizes the required area for the DRBG. The dominating cost in time consumption by this DRBG is caused by two hash function calls. Based on this specification the bit-rate of the streaming output of our random number generator can be approximated as follows:

$$\text{Bit-rate} = \frac{\text{Output bits/hash}}{\text{Number of hash calls}} * \frac{\text{Clock frequency}}{\text{Clocks/hash}} = \frac{256}{2} * \frac{200\text{MHz}}{64} = 400\text{Mb/s}$$

Another parameter that is important for the DRBG is the time that is allowed to elapse before the DRBG needs to be reseeded (hence the SRAM needs to be repowered). After this time an attacker

would be able to start predicting the output bits if the DRBG is not reseeded. According to [1] the DRBG is allowed to produce at most 2^{48} packets of 2^{19} bits before it needs to be reseeded. Converting this requirement into the time before reseeding leads to the following:

$$\text{Time before reseeding} = \frac{\text{Number of output bits}}{\text{bit-rate}} = \frac{2^{48} * 2^{19}}{400\text{Mb/s}} = 11699 \text{ years}$$

From this calculation it becomes clear that our random number generator requires virtually no reseeding and can hence continually produce random bits at a rate of 400Mb/s after the first random seed has been derived.

Finally, in table 6 a rough estimation of the required area in GE (Gate Equivalent) for our implementation can be found. This estimation is based on a non-optimized design (and re-use of the SHA-256 function). It is possible to decrease the required footprint of the design if necessary (e.g., for resource constrained environments).

Table 6. Area estimation of random number generator.

Component	Gate Equivalent
2kB of SRAM	25k
SHA-256 hash	20k
DRBG	10k
Total	**55k**

6 Conclusions

In this paper we have presented the construction and specifications for a random number generator, which is based on the combination of extracting a non-deterministic truly random seed from the noise on the start-up pattern of SRAM memories with a DRBG to convert this seed into streaming of random bits.

The extraction of the physical randomness from SRAM start-up patterns is based on min-entropy calculations that were performed on several different types of SRAM memory. Results from these calculations, as presented in section 3, show that it is important to consider that the random number generator should be able to operate under different circumstances when extracting randomness. Therefore, we have examined the min-entropy present in noise at different temperatures. This has shown that it is possible that under extreme ambient temperatures the amount of randomness present in the SRAM noise decreases. These results are used in our design of the random number generator, which uses an amount of SRAM that provides sufficient entropy for our construction at all measured temperatures.

Based on the results from randomness tests that were performed on seeds derived from 500.000 measurements of an SRAM memory, as presented in section 4, we conclude that it is possible to extract truly random seeds from SRAM noise using a conditioning algorithm.

Combining the extracted randomness with the DRBG, as described in section 5 and [1], results in our random number generator, which is FIPS 140-3 compliant. Furthermore, based on the information from section 5 we can conclude that our implementation of the random number generator is very efficient regarding both resources (area) as well as timing (considering the short time required to derive a truly random seed, after which a constant stream of random bits can be generated at 400Mb/s).

Acknowledgements

This work has been supported in part by the European Commission through the FP7 programme under contract 238811 UNIQUE.

References

1. E. Barker, J. Kelsey, *NIST Special Publication 800-90: Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised)*, March 2007, NIST.
2. J. Daugman, *The importance of being random: statistical principles of iris recognition*, Pattern Recognition, pp. 279-291, Elsevier, 2003.
3. Y. Gao and I. Kontoyiannis and E. Bienenstock, *Estimating the Entropy of Binary Time Series: Methodology, Some Theory and a Simulation Study*, from Entropy, volume 10(2), pp. 71-99, 2008.
4. B. Gassend, D. E. Clarke, M. van Dijk, and S. Devadas, *Silicon Physical Random Functions*, in Vijayalakshmi Atluri editor, Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, pp. 148–160, ACM, 2002.
5. J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, *FPGA Intrinsic PUFs and Their Use for IP Protection*, in Pascal Paillier and Ingrid Verbauwhede, editors, Cryptographic Hardware and Embedded Systems CHES 2007, LNCS volume 4727, pp. 63–80, Springer-Verlag, 2007.
6. D.E. Holcomb, W.P. Burleson, K. Fu, *Initial SRAM state as a Fingerprint and Source of True Random Numbers for RFID Tags*, Conference on RFID Security, 2007.
7. D.E. Holcomb, W.P. Burleson, K. Fu, *Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers*, IEEE Transactions on Computers, 2009.
8. S. S. Kumar, J. Guajardo, R. Maes, G.-J. Schrijen, and P. Tuyls, *The Butterfly PUF: Protecting IP on every FPGA*, in Mohammed Tehranipoor and Jim Plusquellic, editors, IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2008, pp. 67–70, IEEE Computer Society, 2008.
9. J. W. Lee, D. Lim, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas, *A Technique to Build a Secret Key in Integrated Circuits for Identification and Authentication Applications*, in Proceedings of the IEEE VLSI Circuits Symposium, pp. 176–179, 2004.
10. R. Maes, P. Tuyls, and I. Verbauwhede, *Intrinsic PUFs from Flip-flops on Reconfigurable Devices*, In 3rd Benelux Workshop on Information and System Security (WISSec 2008), 17 pages, 2008.
11. A. Maiti, R. Nagesh, A. Reddy, P. Schaumont, *Physical unclonable function and true random number generator: a compact and scalable implementation*, ACM Great Lakes Symposium on VLSI, 2009.
12. T. Moro, Y. Saitoh, J. Hori, T. Kiryu, *Generation of physical random number using the lowest bit of an A-D converter*, Electronics and Communications in Japan (Part III: Fundamental Electronic Science), Volume 89, Issue 6, pages 1321, June 2006.
13. A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, S. Vo, *NIST Special Publication 800-22: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, April 2010, NIST.
14. C. W. Odonnell, G. E. Suh, S. Devadas, *PUF-based random number generation*, in MIT CSAIL CSG Technical Memo 481, 2004.
15. R. S. Pappu, *Physical one-way functions*, PhD. Thesis, Massachusetts Institute of Technology, March 2001.
16. C. S. Petrie, J. A. Connelly, *A Noise-Based IC Random Number Generator for Applications in Cryptography*, IEEE Transactions on Circuits and Systems: Fundamental Theory, Vol. 47, No. 5, May 2000.
17. G. Selimis, M. Konijnenburg, M. Ashouei, J. Huisken, H. de Groot, V. van der Leest, G.J. Schrijen, M. van Hulst, and P. Tuyls, *Evaluation of use of 90nm 6T-SRAM as a PUF for secure key generation in a wireless communication system*, IEEE International Symposium on Circuits and Systems (ISCAS), May 2011.
18. F. Willems, Y. Shtarkov, T. Tjalkens, *Context Tree Weighting: Basic Properties*, IEEE Trans. Inform. Theory 1995, volume 41, pp. 653–664, 1995.
19. F. Willems, Y. Shtarkov, T. Tjalkens, *Context Weighting for General Finite-Context Sources*, IEEE Trans. Inform. Theory 1996, volume 42, pp. 1514–1520, 1996.
20. Federal Information Processing Standards Publication, *FIPS PUB 180-3: Secure Hash Standard (SHS)*, Information Technology Laboratory National Institute of Standards and Technology, Gaithersburg, MD 20899-8900, October 2008.
21. Federal Information Processing Standards Publication, *FIPS 140-3: Security Requirements for Cryptographic Modules, Annex A: Approved Security Functions for FIPS PUB 140-3*, Information Technology Laboratory National Institute of Standards and Technology Gaithersburg, MD 20899-8930, July 2009, Draft

A Test results of NIST tests

This appendix contains more information on the test results from the NIST tests, as performed on data sets 1, 2, and 3. In figure 4 the minimal values of p' for the tests performed on set 1 can be found. The line in this figure indicates the threshold above which these values have to be in order for the test to pass. Most of the tests from the NIST suite are performed on the entire data set of 125 strings. For these tests, the following equation is used to determine the threshold:

$$p' \geq p - 3\sqrt{\frac{p(1-p)}{n}} = 0.99 - 3\sqrt{\frac{0.99 \times 0.01}{125}} = 0.9633 \tag{8}$$

However, the random excursion and random excursion variant tests are performed multiple times on different subsets of this input. The NIST suite selects different subsets consisting of 78 strings each for performing these tests, which leads to the following threshold:

$$p' \geq p - 3\sqrt{\frac{p(1-p)}{n}} = 0.99 - 3\sqrt{\frac{0.99 \times 0.01}{78}} = 0.9562 \tag{9}$$

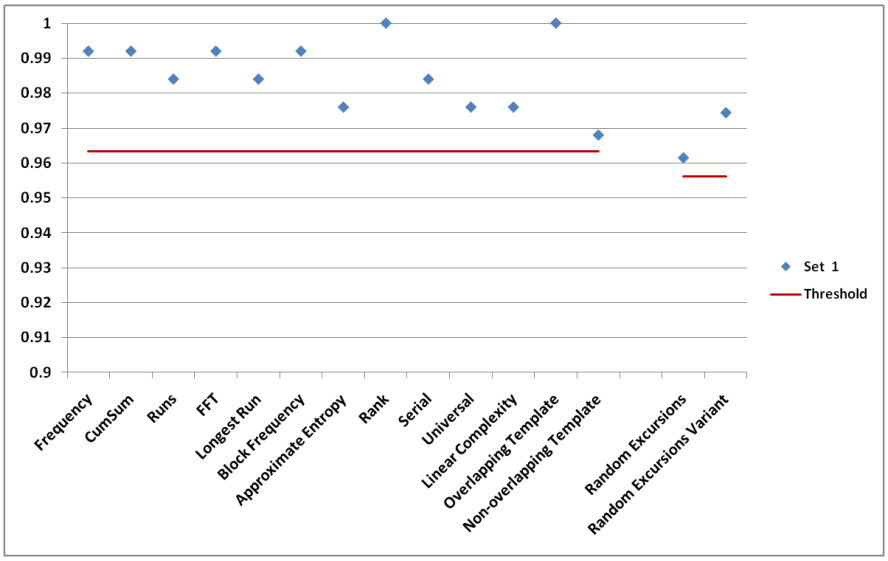

Fig. 4. NIST results from data set 1.

Figure 4 shows that all tests have a minimum value of p', which is higher than their respective threshold. Since the minimal P-value found in during tests on this data set is 0.000924 (which is higher than the threshold for P-value: 0.0001), all NIST tests on data set 1 have been passed.

On data set 2 a subset of the NIST suite has been performed, because the length of the input strings is insufficient for some of the tests. For this data set the following equation is used for calculating the threshold:

$$p' \geq p - 3\sqrt{\frac{p(1-p)}{n}} = 0.99 - 3\sqrt{\frac{0.99 \times 0.01}{250}} = 0.9711 \tag{10}$$

Figure 5 shows the minimal values of p' that were found during the tests as well as the corresponding threshold. Furthermore, the lowest P-value that was observed for this data set was 0.040108, which is more than 0.0001. Therefore, all tests performed on data set 2 have passed.

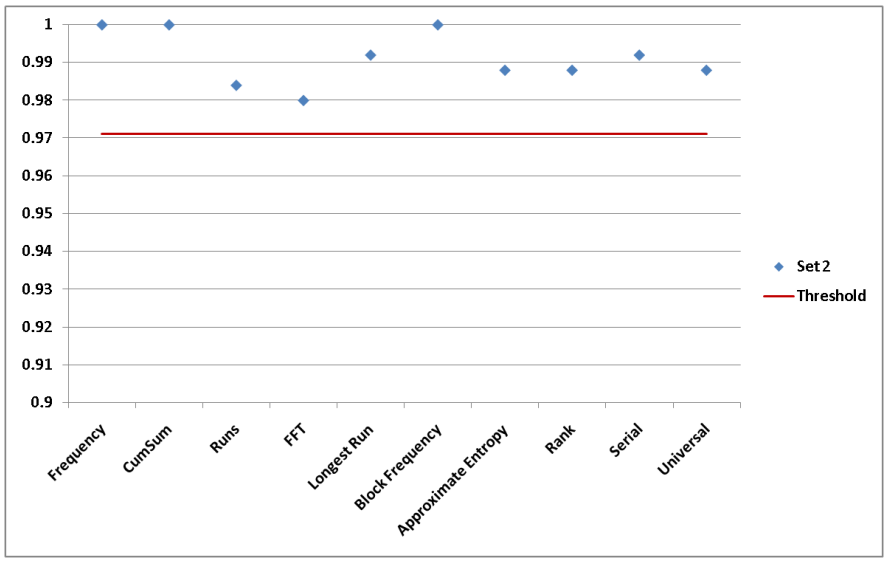

Fig. 5. NIST results from data set 2.

Finally, the results for data set 3 can be found in figure 6. For this set the threshold is:

$$p' \geq p - 3\sqrt{\frac{p(1-p)}{n}} = 0.99 - 3\sqrt{\frac{0.99 \times 0.01}{1000}} = 0.9806 \tag{11}$$

Combining the results from the figure with the fact that the lowest P-value found during these tests was 0.048093, we can conclude that all tests have passed here as well.

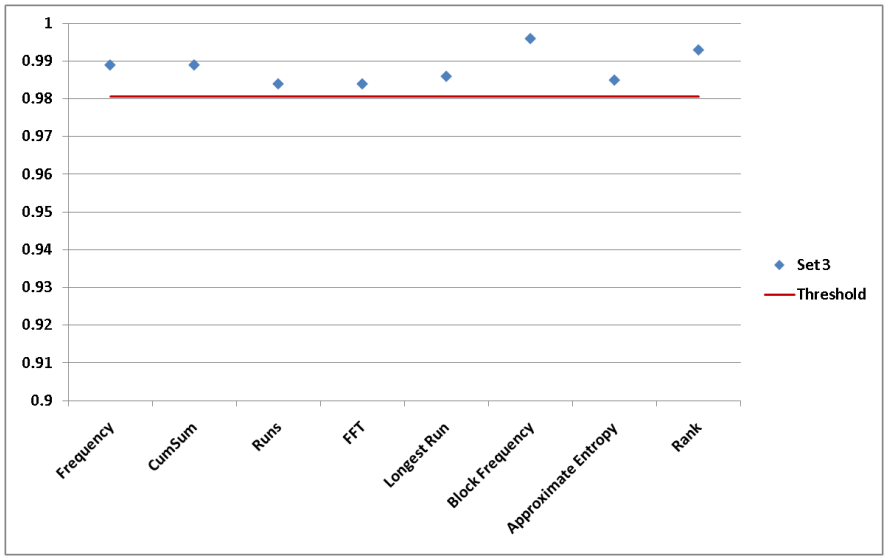

Fig. 6. NIST results from data set 3.