PUF-Based Random Number Generation

Charles W. O'Donnell, G. Edward Suh, and Srinivas Devadas
Computer Science and Artificial Intelligence Laboratory (CSAIL)
Massachusetts Institute of Technology
Cambridge, MA 02139
{cwo,suh,devadas}@mit.edu

ABSTRACT

From security to randomized algorithms, there are many existing problems whose solutions are fundamentally based on the assumption that intrinsically pure random number sources exist. Pseudorandom number generators can imitate randomness sufficiently well for most applications, however, they still rely on some secret seed input. Hardware random number generators attempt to extract randomness directly from complex physical systems. In this way they create random outputs without requiring any seed inputs. In this paper we describe how to use Physical Random Functions (or Physical Unclonable Functions, PUFs) to create a candidate hardware random number generator. We present a short argument supporting the tenability of our methods and provide a brief evaluation of the "randomness" of numbers generated using PUFs through a series of statistical tests. These tests show promising possibilities for the use of PUFs in hardware random number generation.

1. INTRODUCTION

Random numbers are essential ingredients in a great number of solutions in computer science. Randomized algorithms require a random source to guarantee computational complexity bounds and sampling methods often require randomness to accurately represent the information they are surveying. However, one of the most important uses of random numbers comes from cryptographic computer security protocols and algorithms.

Cryptographic applications use random numbers to generate encryption keys, create initial parameter values, and to introduce random nonces into protocols and padding schemes. In most cases a these random numbers come from a Pseudorandom Number Generator (PRNG) which is a deterministic software algorithm which *imitates* randomness. A PRNG takes an input string of bits, or a bit-vector, and generates a longer output bit-vector which "appears" random. Although there are many definitions of what it means for a bit-vector to "appear" random, one general rule requires that a pseudorandom number is indistinguishable from a "truly" random number. That is, given a random number and a pseudorandom number, it is computationally intractable to execute an algorithm which could identify the pseudorandom number. This is related to another important criteria which requires that every "next" output bit from the random number generator must be unpredictable or uncorrelated to all prior bits.

While many pseudorandom number generators exist, not all of them satisfy the indistinguishability requirement and are therefore not cryptographically sound. For example, the ISO C rand() function would not create "good" cryptographic keys. It can be shown, however, that any candidate *one-way function* can be used directly as pseudorandom number generator that is cryptographically sound. Therefore, algorithms such as SHA-1 [5] or RSA Laboratories' BSAFE [1] are commonly used as a PRNG.

Unfortunately, PRNGs suffer from two major security disadvantages. First, PRNGs require some input which deterministically governs the output. To securely use a PRNG, this input (the "seed") must be kept *secret*. Second, PRNGs can only generate a fixed number bits before they cycle and repeat themselves. For applications that require extremely long periods of random bits this can be a problem.

Hardware random number generators (HRNG) do not suffer from these two issues since they generate aperiodic "random" output without the need of input. To do this they generate output bits by exploiting inherent unpredictability in complex physical systems and processes. Because of this, the security of any HRNG is directly tied to the infeasibility of modeling and imitating the underlying physical system.

Although hardware generators are quite difficult to design, and even more difficult to validate, a number of candidate HRNG designs have been developed [3, 6, 8, 12].

In this paper we consider the implementation of an HRNG whose underlying physical system is based upon Physical Random Functions. We present one candidate HRNG design and analyze its potential.

Section 2 introduces Physical Random Functions and describes their typical use and characteristics. In Section 3 we specify our candidate HRNG and illustrate how it makes use of Physical Random Functions. Section 4 gives a brief analysis of the "randomness" of our generator and Section 5 concludes.

2. PHYSICAL RANDOM FUNCTIONS

A Physical Random Function (also called a Physical Unclonable Function, or PUF) is a function that maps a set of challenges to a set of responses based on an intractably complex physical system. Under what we will call "normal" operation this is an arbitrary ("random") but static mapping between challenges and responses. We will later discuss the implications of non-static mappings. The physical random function can *only* be evaluated using the physical system, and is unique for each physical instance. While PUFs can be implemented with various physical systems, we only consider the use of silicon PUFs (SPUFs) in this paper. SPUFs generate their responses based on the hidden timing and delay information of integrated circuits, and will now be described.

Even with identical layout masks, the variations in the manufacturing process cause significant delay differences among different ICs. Because the delay variations are random and practically impossible to predict for a given IC, we can extract secrets unique to each IC by measuring or comparing delays at a fine resolution.

Figure 1 illustrates the SPUF delay circuit used in this paper. While this particular design is used to demonstrate the technology, note that many other designs are possible. The circuit has a

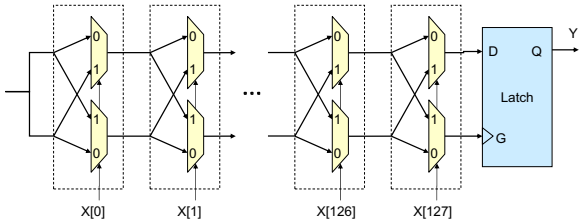

Figure 1: A silicon PUF delay circuit.

multiple-bit input X and computes a 1-bit output Y by measuring the relative delay difference between two paths with the same layout length. The input bits determine the delay paths by controlling the MUXes, which either switch or pass through the top and bottom signals. To evaluate the output for a particular input, a rising signal is given to both top and bottom paths at the same time, the two signals race through the two delay paths, and the latch at the end measures which signal is faster. The output is *one* if the top signal is faster, and *zero* if the bottom signal is faster.

This PUF delay circuit with 64 stages has been fabricated and tested in TSMC's 0.18μm, single-poly, 6-level metal process [9]. Experimental results show that multiple measurements of a single challenge on the same chip has a different response with a probability of 0.7%. Environmental variations such as temperature change or voltage aberrations can increase this probability further. Changing the temperature from 20 to 70 Celsius causes noise of 4.8% while varying the regulated voltage by $\pm 2\%$ causes noise of 3.7%. Higher changes in temperature and voltage can increase this noise to upward of 9%.

Experiments also show that approximately 0.1% of all challenges do not return a consistent response at all [10]. These meta-stable challenges generate responses which can vary unpredictably (one can say they create a non-static mapping, or have 50% variation). Environmental changes are much more influential to these meta-stable challenges since small changes in voltage or temperature can cause a challenge's response to no longer behave in an unpredictable fashion.

3. PUF-BASED RANDOM NUMBER GENERATION

Instead of using the PUF circuit for "normal" operation, we will focus on the meta-stable PUF challenges mentioned in Section 2. We discuss here our candidate methodology for generating 32-bit random numbers using these unpredictable challenges to extract randomness from the PUF physical system. For our purposes, we define an "unpredictable challenge" to be one which, when applied to the PUF, will result in a one with probability $0.5 \pm \epsilon$.

Figure 2 depicts a high-level view of how our HRNG design operates. Our design accepts a single incoming request for a random output and produces an output using an iterative process which discovers a challenge which gives unpredictable results. Once a suitable challenge is found, a post-processing step is applied to remove bias and extract randomness from the bit ordering.

In our design an unpredictable input challenge is saved in a local register, however we cannot simply choose one unpredictable challenge at bootup and use it for every request. As we saw in Section 2, the response of a challenge is not only dependent on the physical system, but is also dependent on environmental characteristics such as temperature and voltage. Therefore every time a random number request is made, we must confirm that the challenge we are using is still "unpredictable" (although it may often be the case that the previous challenge used is still unpredictable).

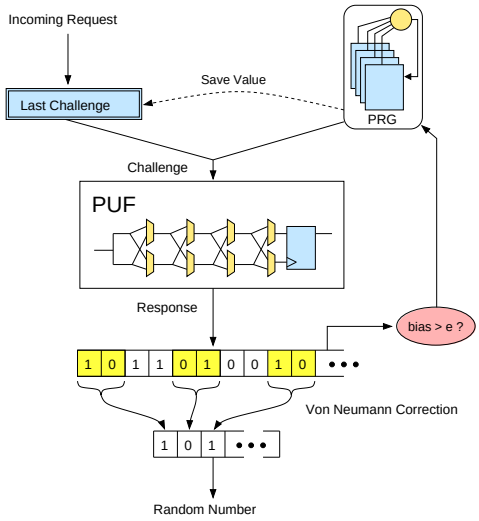

Figure 2: PUF random number generation

This confirmation is done by examining the output response bits to see if they *appear* to produce ones with a probability of $0.5 \pm \epsilon$. Specifically for this candidate design, we feed the challenge to the PUF N times (generate N bits) and check to make sure that $\frac{N}{2} \pm \varepsilon$ of the responses resulted in *one*, while the remaining were *zero*. If the challenge is found to have a bias, we generate these N bits $(M-1)$ more times using this same challenge. If the challenge fails this bias test all M times, the challenge is fed into a pseudo-random number generator to create a new challenge to be tested. A simple PRNG can be implemented as a linear feedback shift register which saves the next value directly into a register. We note that this challenge is *not* a seed and does not need to be kept secret. Also, the PRNG which generates the next challenge does not need to be cryptographically secure, but is only used to uniformly select possible challenges.

Finally, we derive our final 32 random bits from these N bits by applying a Von Neumann corrector. Although we artificially require our output distribution to have $\frac{N}{2} \pm \varepsilon$ ones, we use Von Neumann correction to extract randomness from the temporal *ordering* of ones and zeros. In this method we scan the random sequence from left to right reading successive pairs of bits. If the two bits in a pair differ, we use the first bit as a final random bit, if the two bits are the same, we simply move to the next pair. In this way we parse the N bits until 32 random bits have been determined. In the rare chance 32 bits are not produced, we rerun the challenge through the PUF to get N new bits.

4. ANALYSIS

To demonstrate that a PUF can be used as a random source for cryptography, we must analyze the complexity of modeling or imitating a PUF circuit and look at the inherent unpredictability of these "unpredictable responses." Supplemental to this, we can bolster these claims by performing a number of statistical tests to determine randomness as suggested by NIST [11, 7]. These tests are not absolute, however they give a good sense of the performance of a given HRNG (even a truly random number may fail some of the tests). There is also a great amount of further work focused on the problem of analyzing random number generators [2, 13, 14, 4].

We note also that this analysis is brief, and it is certainly possible to discuss the theoretical and philosophical "randomness" of our

candidate methodology at a much deeper level.

4.1 Unpredictable PUF Challenges

Fundamentally, our random number generator depends on the unpredictability of small number of meta-stable challenges. We argue that these challenges are unpredictable since all SPUF challenges are based on the hidden and unknown timing variations inherent in chip manufacturing. Any attempt by an adversary to determine the output of a given PUF challenge is highly likely to change the PUF circuit itself [10]. Further complicating matters for an adversary, these unpredictable challenges change quite frequently due to tiny environmental variations. When running experiments which continually generated random bits, we found that any given unpredictable challenge only continued to remain unpredictable for a few seconds at most.

We qualified these challenges as unpredictable only when they would pass a simple bias test. While an unpredictable challenge will not always pass this test, the likelihood of it passing is

$$\sum_{i=0}^{2\varepsilon} \binom{N}{\frac{N}{2} - \varepsilon + i} \left(\frac{1}{2}\right)^N.$$

If $N = 1,000$ and $\varepsilon = 50$ (that is, $\epsilon = 0.05$), then the probability of the challenge passing the bias test is greater than 0.998. By re-running this test M times we increased our chances greatly of determining whether a particular challenge was acting appropriately unpredictable.

4.2 Statistical Tests

To generate a stream of random numbers, we implemented the methodology discussed in Section 3 using an implementation of the AEGIS secure processor [15]. This processor contains a PUF circuit which is accessible through software running on the chip. Therefore we implemented the Von Neumann Correction and PRNG in software and used the chip's PUF circuit to generate responses. For the following analysis, we used constants of $N = 1,000$, $M = 100$, and $\varepsilon = 50$.

The NIST Statistical Test Suite is a set of algorithmic tests which attempt to identify sequences of binary numbers which do not behave in a truly random manner. To do this, these tests derive a $p\text{-}value$ for every sequence of bits, which is, basically, the probability that the given sequence could have been generated by running a truly random number generator once. Each test "passes" if the $p\text{-}value$ is greater than some fixed confidence level. A thorough explanation of these testing algorithms can be found in a NIST Special Publication [11].

The statistical test suite also separates a given input string into many smaller substrings and performs the tests on each of these strings. Since it is possible for a truly random number generator to fail a given test *sometimes*, it is useful to discuss success in terms of the proportion of successful tests.

Table 1 shows the lowest proportion of successful tests found after running the test suite many times on hundreds of megabytes of our random data (that is, the worst-case numbers we encountered). The specific input parameters used for each test are also listed. We can see that the proportion of successful tests is high enough to consider this a reasonably good random number generator.

Statistical test	Block/Template length	Lowest success ratio
Frequency	-	100%
Frequency within Blocks	128	96%
Cumulative Sums	-	100%
Runs	-	94%
Longest Run within Blocks	-	98%
Binary Rank	-	98%
FFT	-	98%
Non-periodic Templates	9	94%
Maurer's Universal Test	7	100%
Approximate Entropy	10	96%
Random Excursions	-	93.5%
Serial	16	99%
Linear Complexity	500	100%

Table 1: NIST Statistical Test Suite success ratio

5. CONCLUSIONS

We have put forward one candidate methodology which uses Silicon Physical Random Functions to generate random numbers without the need of a secret seed. This hardware random number generator is based on the meta-stability inherent in SPUFs and has been shown to produce outputs which pass most statistical tests for randomness. This suggests that PUF-based random number generation is a cheap and viable alternative to more expensive and complicated hardware random number generation.

6. REFERENCES

[1] R. W. Baldwin. Preliminary analysis of the BSAFE 3.x pseudorandom number generators. Technical Report 8, RSA Laboratories, 1998.

[2] R. Davies. Hardware random number generators. In 15^{th} *Australian Statistics Conference*, July 2000.

[3] R. B. P. Dept. The Evaluation of Randomness of RPG100 by Using NIST and DIEHARD Tests. Technical report, FDK Corporation, 2003.

[4] D. Eastlake and S. Crocker. RFC 1750: Randomness recommendations for security, Dec. 1994.

[5] D. Eastlake and P. Jones. RFC 3174: US secure hashing algorithm 1 (SHA1), Sept. 2001.

[6] B. Jun and P. Kocher. The Intel Random Number Generator. Cryptography Research Inc. white paper, Apr. 1999.

[7] S. Kim, K. Umeno, and A. Hasegawa. On the NIST Statistical Test Suite for Randomness. In *IEICE Technical Report, Vol. 103, No. 449, pp. 21-27*, 2003.

[8] P. Kohlbrenner and K. Gaj. An embedded true random number generator for fpgas. In *FPGA '04: Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 71–78. ACM Press, 2004.

[9] J.-W. Lee, D. Lim, B. Gassend, E. G. Suh, M. van Dijk, and S. Devadas. A Technique to Build a Secret Key in Integrated Circuits with Identification and Authentication Applications. In *Proceedings of the IEEE VLSI Circuits Symposium*, June 2004.

[10] D. Lim. Extracting Secret Keys from Integrated Circuits. Master's thesis, Massachusetts Institute of Technology, May 2004.

[11] NIST Special Publication 800-22. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Information Technology Laboratory of the National Institute of Standards and Technology, May 2000.

[12] C. Petrie and J. Connelly. A Noise-based IC Random Number Generator for Applications in Cryptography. *IEEE TCAS II*, 46(1):56–62, Jan. 2000.

[13] W. Schindler and W. Killmann. Evaluation criteria for true (physical) random number generators used in cryptographic applications. In *CHES '02: Revised Papers from the 4th*

International Workshop on Cryptographic Hardware and Embedded Systems, pages 431–449. Springer-Verlag, 2003.

[14] Securitytechnet random number generator references. `http://www.securitytechnet.com/crypto/algorithm/random.html`.

[15] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. Technical report, MIT CSAIL CSG Technical Memo 483, November 2004.