An Investigation of Sources of Randomness Within Discrete Gaussian Sampling

Séamus Brannigan[1], Neil Smyth[1], Tobias Oder[2], Felipe Valencia[3], Elizabeth O'Sullivan[1], Tim Güneysu[4] and Francesco Regazzoni[3]

[1] Centre for Secure Information Technologies (CSIT), Queen's University Belfast, UK
[2] Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
[3] Advanced Learning and Research Institute, Università della Svizzera Italiana, Switzerland
[4] Universität Bremen, Germany

Abstract. This paper presents a performance and statistical analysis of random number generators and discrete Gaussian samplers implemented in software. Most Lattice-based cryptographic schemes utilise discrete Gaussian sampling and will require a quality random source. We examine a range of candidates for this purpose, including NIST DRBGs, stream ciphers and well-known PRNGs. The performance of these random sources is analysed within 64-bit implementations of Bernoulli, CDT and Ziggurat sampling. In addition we perform initial statistical testing of these samplers and include an investigation into improper seeding issues and their effect on the Gaussian samplers. Of the NIST approved Deterministic Random Bit Generators (DRBG), the AES based CTR-DRBG produced the best balanced performance in our tests.

1 Introduction

Lattice-Based Cryptography is currently one of the leading contenders for Post-Quantum Cryptography. Cryptographic constructions for encryption, digital signatures and key exchange based on hard problems in Lattices are likely to be submitted to the current NIST call for Post-Quantum algorithms [1]. Major efforts are in place to produce practical implementations in both hardware and software of the available and future algorithms. It is essential that these implementations are examined by the cryptographic and security engineering communities to build a level of confidence in the algorithms over the course of the submission and evaluation periods. The EU H2020 SAFEcrypto project is currently developing an open source library containing implementations of several Lattice-based schemes and a phased release process over the next two years is planned.

Many modern Lattice-based cryptosystems contain discrete Gaussian sampling as a core component. Discrete Gaussian sampling aids theoretical proofs and enables the design of constructions with more compact key sizes. While this component may not be the main bottleneck within constructions, it is nevertheless a significant component and proposals for efficient sampling techniques, particularly for constrained devices, is an active area of research. Several sampling techniques have been analysed in the literature for this purpose including Bernoulli sampling [2,3], Knuth-Yao sampling [4], CDT [5] and the Ziggurat sampler [6,7]. Recently, Saarinen [8] proposed that the precision of Gaussian samplers may be reduced to 64-bit with negligible effect on security for most implementations. Such a proposition is extremely desirable as this results in significant savings in speed and compactness. Implementations and evaluations based upon this proposal have been investigated in hardware [9]. However, recommendations for software will not necessarily be the same. The constraints on memory, communication bandwidth, power consumption in hardware implementations often necessitate a different architectural approach than software. Open software implementations of the Bernoulli sampling method and CDT sampling have been provided by Oder et al. [10] and Saarinen [11], respectively. This paper examines 64-bit versions of these samplers together with a 64-bit C implementation of discrete Gaussian sampling using the Ziggurat method. This new Ziggurat implementation demonstrates performance comparable to CDT sampling but uses significantly less memory. All implementations are designed such that they offer full flexibility for a range of parameters with respect to tailcut, standard deviation and number

of rectangles (Ziggurat). We have not considered Knuth-Yao in this study as the implementation we have was specifically designed for hardware modelling and is not an optimal software design, therefore we feel that a comparison with software-specific designs would be misleading.

To date, most of the research in discrete Gaussian sampling within the context of Lattice-based cryptography has focused on the design and optimisation for performance enhancements, assuming the existence of a source of true random number generation. In practice, many software implementations will use pseudo random number generators (PRNGs). Therefore, this paper examines the effect of several PRNG sources that are likely to be chosen for this purpose. The performance of the discrete Gaussian sampling component is affected by the performance of the selected PRNG, and subsequently the lattice primitive is also affected. Of course, implementations can be aggressively optimised to produce optimal performance in terms of speed on target platforms such as ARM cortex or AVX. However, there is also a need for generic implementations that will suit a wide variety of platforms and can support flexible parameter choices. The algorithms analysed in this work are general purpose implementations using non-aggressive compiler optimisations.

A number of NIST approved DRBG schemes are examined due to their prevalence and acceptance by the security community. An AES-based CTR-DRBG is examined because of its widespread availability as a co-processor in many microcontrollers, and a SHA-2 based HASH-DRBG is examined as it is considered to have the best performance of any hash-based NIST DRBG. In addition to NIST approved schemes, the stream cipher ChaCha20 [12] is chosen because it has been used in other Lattice-based schemes [13] and is an increasingly popular PRNG, as demonstrated by its selection as the replacement for RC4 in Google's TLS [14]. Also selected for comparative purposes is ChaCha20's predecessor, Salsa20, which is still popular in many software cryptographic libraries. The less popular but long-standing ISAAC CSPRNG is also analysed due to its continued resilience to attack and high performance. A 64-bit implementation of the KISS PRNG is included, as this PRNG accompanied the original Marsaglia and Tsang Ziggurat implementation [6]. Finally we include the standard RNG functions found on a typical Linux OS, namely *rand(), random()* and access to the special files */dev/random* and */dev/urandom*. The performance of these PRNGs combined with 64-bit implementations of Ziggurat, Bernoulli, and CDT samplers is examined on an Intel i7 6700 CPU at 3.4GHz.

This paper is organised as follows. In Section 2 we summarise Gaussian sampling and the various RNGs that have been selected. Section 3 discusses the various software implementation decisions and design goals, potential security and performance issues and the importance of testing RNGs. Section 4 contains a PRNG performance evaluation. Section 5 contains our initial statistical analysis of the PRNGs within the context of discrete Gaussian sampling. A discussion and conclusions are drawn in section 6.

2 Background

2.1 Gaussian Sampling

A number of cryptosystems in Lattice-based cryptography rely on sampling from the discrete Gaussian distribution, namely over the integers, such that the probability, p, of a number, x, being sampled is proportional to the Gaussian function:

$$p_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-x^2/2\sigma^2}. \tag{1}$$

Methods for generating discrete Gaussian samples often take the form of inversion methods or rejection methods. Typically, inversion methods involve inverting cumulative distribution functions. This is usually achieved by inverting a closed formula or inverting an approximation of it. Most common inversion methods for discrete random variables take the form of table lookup methods, the advantage of this is that they can be constant time but the table size can be large. Rejection methods are useful when inversion is not easily computed, or when the size of the table lookup becomes too

large. The basic method of 'rejection' sampling a distribution is to uniformly sample two numbers, x and y. If $y \leq p_\sigma(x)$, then x is returned as the sampled variable. In other words, x is rejected with probability determined by the distribution.

In Section 5 we discuss the effects of the underlying RNG required by Gaussian sampling schemes. In particular we discuss the effects of various RNG choices on the Bernoulli, CDT and Ziggurat samplers. We also illustrate the vulnerabilities that can be exposed in the cryptosystem by improper seeding.

2.2 Random Number Generators

This Section provides an overview of the selected RNGs in our tests including NIST DRBGs, stream-based CSPRNGs, high-performance PRNGs and system P/RNGs.

NIST DRBGs

CTR-DRBG

CTR-DRBG is defined by NIST [15] as a block cipher Deterministic Random Bit Generator (DRBG) based on a NIST approved block cipher. We have selected AES-256 in order to achieve the maximum security strength of 256 bits. In practice, this DRBG is best described as AES Counter mode, with an external entropy source used in the generation of keys and the initial counter value. This DRBG must be periodically reseeded to prevent the sequence of random bytes being repeated. Also, reseeding must occur at a minimum of every 2^{19} bits as a means of resistance against backtracking [15].

We have evaluated two versions of a publicly available implementation of AES [16]. One version is a generic C implementation using no assembler optimisation, the other utilises Intel AES-NI hardware acceleration. We have compiled this implementation using full loop unrolling, static lookup tables and maximal use of pre-computed round tables. However as this implementation makes significant use of lookup tables it is considered vulnerable to cache timing attacks, therefore the AES-NI version is recommended as it avoids such attacks and improves performance.

Hash-DRBG

Hash-DRBG is defined by NIST [15] as a hash-based DRBG based on a NIST approved hash function. This DRBG relies upon an external entropy source for seeding purposes and must be periodically reseeded. Its security strength is dependent upon the underlying hash function. We have selected SHA-256 and SHA-512 as the underlying hash functions for our analysis as they are specified for use in the Lattice-based cryptosystem signature scheme BLISS-B [17,18], both of which have a security strength of 256 bits and are publicly available [19].

We have also considered other hash functions such as SHA-3, BLAKE2 and Whirlpool but will elaborate on their use in the coming months. SHA-3 is of significance as it is considered to be highly efficient in a hardware implementation and SHA-3 instructions are likely to be introduced in future processor architectures, in the same manner that SHA-1 and SHA-256 instruction set extensions were introduced with Intel SHA Extensions and ARMv8 Cryptography Extensions. We have observed that BLAKE2 is highly suited to SIMD extensions and can be significantly accelerated with little effort using GNU C Compiler auto-vectorisation, however this is unlikely to translate to fast Hash-DRBG performance as hashing small messages does not exploit this speed advantage.

Stream CSPRNGs

Salsa20 and ChaCha20

Salsa20 is a stream cipher developed by Daniel J. Bernstein in 2005 and subsequently released into the public domain. It is closely related to the more recent ChaCha20 [20] stream cipher that was released in 2008. These are both pseudorandom functions that can be used as CSPRNG's when

combined with the use of a counter and appropriate seeding material. In this evaluation we are using the publicly available C implementations of Salsa20 [21] and ChaCha20 [22].

CSPRNG variants of both of these stream ciphers have been implemented in various cryptographic libraries. ChaCha20 has received particular attention to date in terms of standards adoption, where it has been drafted as a standard within TLS [14], IKE and IPsec [23]. It is also used as a CSPRNG within the Linux kernel since version 4.8 as a means of generating data for the */dev/urandom* device.

High-Performance PRNGs

ISAAC

ISAAC [24] (Indirection, Shift, Accumulate, Add and Count) is a CSPRNG and stream cipher developed by Bob Jenkins in 1993 and released into the public domain. It has similarities to RC4, maintaining two 256 eight-octet arrays of integers as its state with minimal operations required to generate output data resulting in impressive speed. We are using the author's public domain 64-bit C implementation of ISAAC [25].

KISS

KISS (Keep It Simple Stupid) [26,27] is a pseudorandom number generator that has been developed for performance in terms of speed. Whilst it is not a CSPRNG and should not be implemented in a cryptographic application, it is included in this comparison as it is a component of the reference implementation of the Ziggurat Gaussian sampler, as such it allows us to illustrate any possible issues in an Lattice-based cryptosystem implementation that uses it as a random source.

System P/RNGs

glibc rand() and random()

The implementation of ISO C standard *rand()* is system and compiler dependent. It returns a pseudo-random integer in a specified range and is initialised using a call to *srand()*. If no seed call is made the internal state is initialised using the value 1. In most Linux systems *rand()* is an alias for *random()* which is more complex and less susceptible to state recovery by means of linear algebra, but still utilizes a 32-bit seed which is weak to brute force attack.

```
static unsigned long next = 1;

/* RAND_MAX assumed to be 32767 */
int myrand(void) {
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}

void mysrand(unsigned seed) {
    next = seed;
}
```

Fig. 1: POSIX recommended rand() for compatibility between different machines

The following example shown in Figure 1 is an implementation of *rand()* provided by [28] that uses a linear congruential generator to address the issue of generating identical random number sequences on different machines. Such a random number generator is fast but is easily susceptible to either brute force attack or linear algebra to recover the state and allow generation of the random number sequence.

As an example of state recovery using linear algebra, an attacker can exploit the knowledge that the state x after n calls to *rand()* will be $x + n = (x + n) * 1103515245 + 12345$ and the output will differ by approximately $(n * 1103515245) >> 16 = n * 16838$ (excluding rounding and truncation).

/dev/random and /dev/urandom

The */dev/random* device gathers environmental noise from the system to provide an entropy source that can be used to provide random data. The rate at which environmental noise is generated to produce random data is limited, therefore any C library calls that access this device must be blocking. The */dev/urandom* device is provided as a non-blocking alternative, and as such it utlizes a CSPRNG to supplement the environmental noise and provide random data at higher rates.

3 Implementation

A practical implementation of a software CSPRNG must consider both the security and performance of that CSPRNG depending upon the application. For example, someone creating a PGP key pair to secure their emails for long term security can tolerate a high entropy source with low throughput on a single occassion, therefore they can utilise a random source such as */dev/random*. However, a server application with many connections requires a high throughput that can not be serviced by */dev/random*, instead it must rely upon dedicated hardware RNG peripherals or software CSPRNGs.

It is typical for a software CSPRNG to be structured such that low bandwidth random number generation from high entropy sources is used to seed a high bandwidth CSPRNG. Such a scheme provides random number generation that can be scaled to meet the demands of high bandwidth multi-threaded server applications or low bandwidth client applications.

With these considerations in mind we have developed an interface of a range of entropy sources and PRNGs for the purpose of evaluating their performance and suitability for use within software Lattice-based cryptosystems. It is intended that the resulting implementation will provide a robust means of random number generation for Lattice-based schemes.

3.1 Entropy and Seeding

A critical component of a CSPRNG is the selection of seeding material and when to apply the seeding operation. If an attacker can gain knowledge of the seed used to initialise a CSPRNG this can result in significant vulnerabilities as while the random numbers exhibit all the properties required of randomness, the sequence of numbers is algorithmic and predictable. A simple example of this is a naive implementation of seeding an RNG using the current date/time with a lowest granularity of seconds, as an attacker can guess with high predictability the seed which was used with such a low entropy source. It is therefore imperative that the seed used for a CSPRNG is from a high entropy source.

Many CSPRNG algorithms rely upon some form of counter as a cipher input to produce random data, e.g. NIST CTR-DRBG. It is important that the counter is not allowed to wrap around such that the random number sequence is repeated. To prevent this, the CSPRNG must be re-seeded before the sequence of counter values is exhausted.

3.2 Security versus Performance

In many applications and protocols there exists a hierarchy of keys in which the security strength is variable. Typically, high security is required when generating a key pair for long-term use (i.e. at install time) and lower security is sufficient for ephemeral keys that are short-lived (e.g. when a communication session is initiated). Such applications can be exploited by the cryptosystem to improve performance by means of using more efficient cryptographic processing with lower security strength where it will not compromise the system.

The seeding of all PRNG algorithms follows one of four mechanisms to provide the seed entropy:

1. SEI CERT C secure coding standards [29, see MSC32-C] whereby the seed is composed of the current time scrambled using an XOR of the second and nanosecond variables.
2. Entropy is provided by */dev/random*, where stream reading will block until data is available.
3. Entropy is provided by */dev/urandom*, where stream reading will not block until data is available.
4. Entropy is provided by a source file (*for debugging and testing purposes only*). The test file must contain binary data, once exhausted the file content is repeated.

4 Performance

4.1 Raw Performance

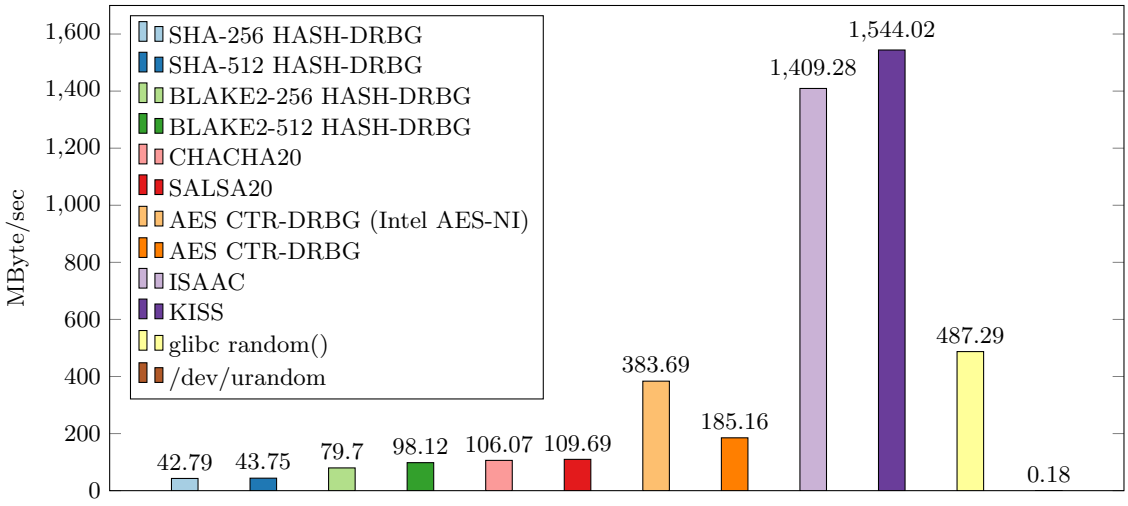


Fig. 2: Performance comparison of PRNG schemes [Intel i7 6700 CPU @ 3.4 GHz] with GNU C Compiler auto-vectorisation *enabled*

The raw performance of all PRNGs is shown in Figure 2 when used in the same software testbench to generate 128 MBytes of random data. KISS and ISAAC have the best performance in terms of speed, achieving more than three times the speed of Intel AES-NI accelerated CTR-DRBG. The glibc *random()* function achieves the third highest speed, while the Salsa20 and ChaCha20 stream ciphers obtain disappointing speeds that are less than 8% of that achieved by ISAAC and less than 30% of that achieved by AES CTR-DRBG. The SHA-2 HASH-DRBGs obtain very low performance of ~3% of that achieved by ISAAC and less than 12% of that achieved by AES CTR-DRBG, indicating that CTR-DRBG should be used in preference to HASH-DRBG if a NIST random number generator is required. The high entropy */dev/urandom* is clearly shown to provide non-practical performance and thus is removed from further testing within samplers.

These figures were obtained using GNU C Compiler (GCC) 5.4.0 with moderate optimisation $-O2$ and auto-vectorisation enabled, i.e. SIMD hardware is exploited where possible by the compiler. The auto-vectorisation feature of GCC is useful for exploiting SIMD instruction sets without resorting to instrinsics or assembler, allowing generic C code to be maintained for use on different processor technologies such as Intel AVX2 or ARM NEON. However, to illustrate the benefits of SIMD instructions

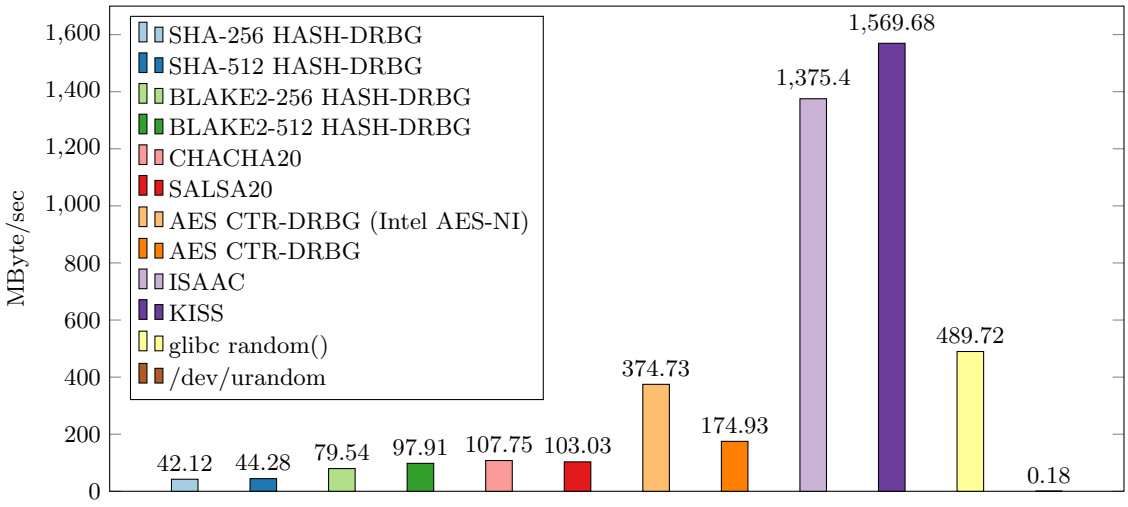


Fig. 3: Performance comparison of PRNG schemes [Intel i7 6700 CPU @ 3.4 GHz] with GNU C Compiler auto-vectorisation *disabled*

we have included a comparison of the various PRNGs with auto-vectorisation disabled in Figure 3. We have used */dev/urandom* as an entropy source for seeding the PRNGs.

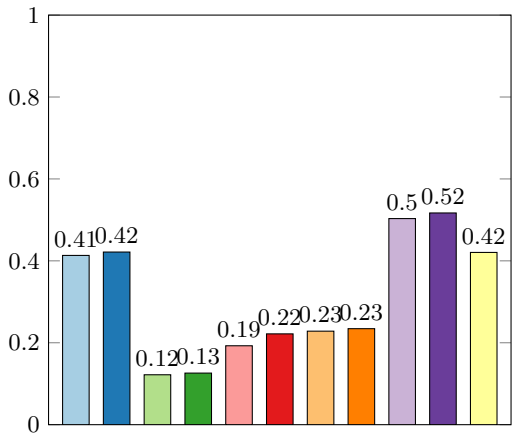
Auto-vectorisation disabled shows us that there is a small performance gain (accounting for measurement errors) from the various RNGs when SIMD instructions are used. None of the RNGs selected utilize an underlying block cipher, stream cipher, hash function or algebra that is particularly suited to SIMD instructions or which the compiler has success in automatically optimising. However, the authors understand that hand-optimised implementations [30] can be obtained for many of these functions.

4.2 Performance with Gaussian Sampling

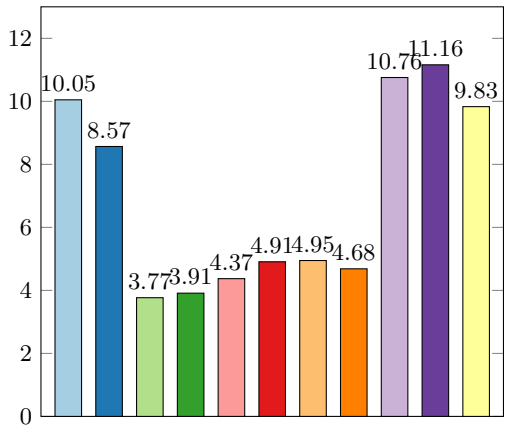
A comparison of the data througput achieved by the three Gaussian samplers, namely Bernoulli, CDT and Ziggurat, using various PRNGs is shown in Figures 4 and 5. The relative performance of the Gaussian sampler directly correlates with the performance of the underlying PRNG, although the overall performance boost is limited by bottlenecks elsewhere within the the Gaussian samplers. For example, while ISAAC is more than 300% faster than AES CTR-DRBG in terms of raw speed, when used within a Gaussian sampler its speed improvement is reduced to 20-30%. While the performance degradation is still significant, the poor performance of Salsa20, ChaCha20 and the Hash-DRBG as PRNGs also effects the Gaussian sampler speed to a lesser extent. Interestingly, hardware accelerated AES consistently provides a marginally lower performance than a software implementation when used within a Gaussian sampler.

As discussed previously, these figures were obtained using GNU C Compiler (GCC) 5.4.0 with moderate optimisation $-O2$, auto-vectorisation enabled (see Figure 4) or disabled (see Figure 5) and we have used */dev/urandom* as an entropy source for seeding the PRNGs.

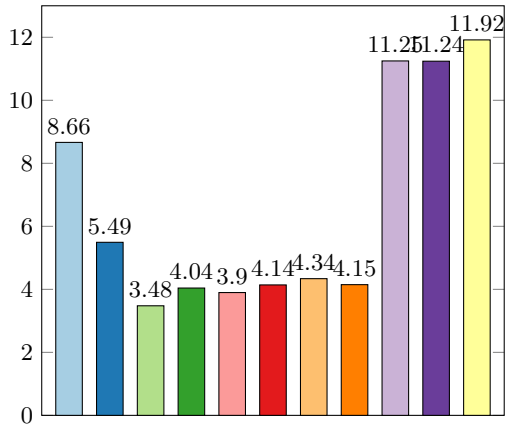
Sampling from a Gaussian distribution is a component within most Lattice-based schemes that requires a significant source of random data, therefore any speed advantage that can be gained from the choice of Gaussian sampler used within that Lattice-based cryptosystem is important. The authors' intend to expand the results obtained here to evaluate the performance of various Gaussian samplers as used within a number of Lattice-based schemes and the suitablility of these Gaussian samplers to multithreaded applications.



(a) Bernoulli sampling (MByte/sec)



(b) CDT sampling (MByte/sec)



(c) Ziggurat sampling (MByte/sec)

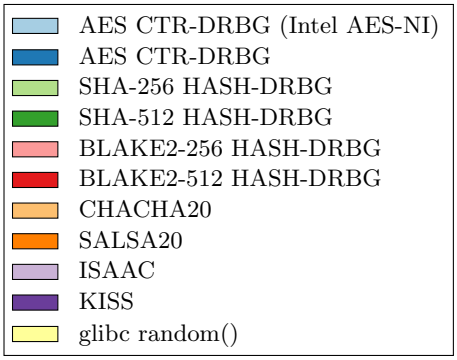
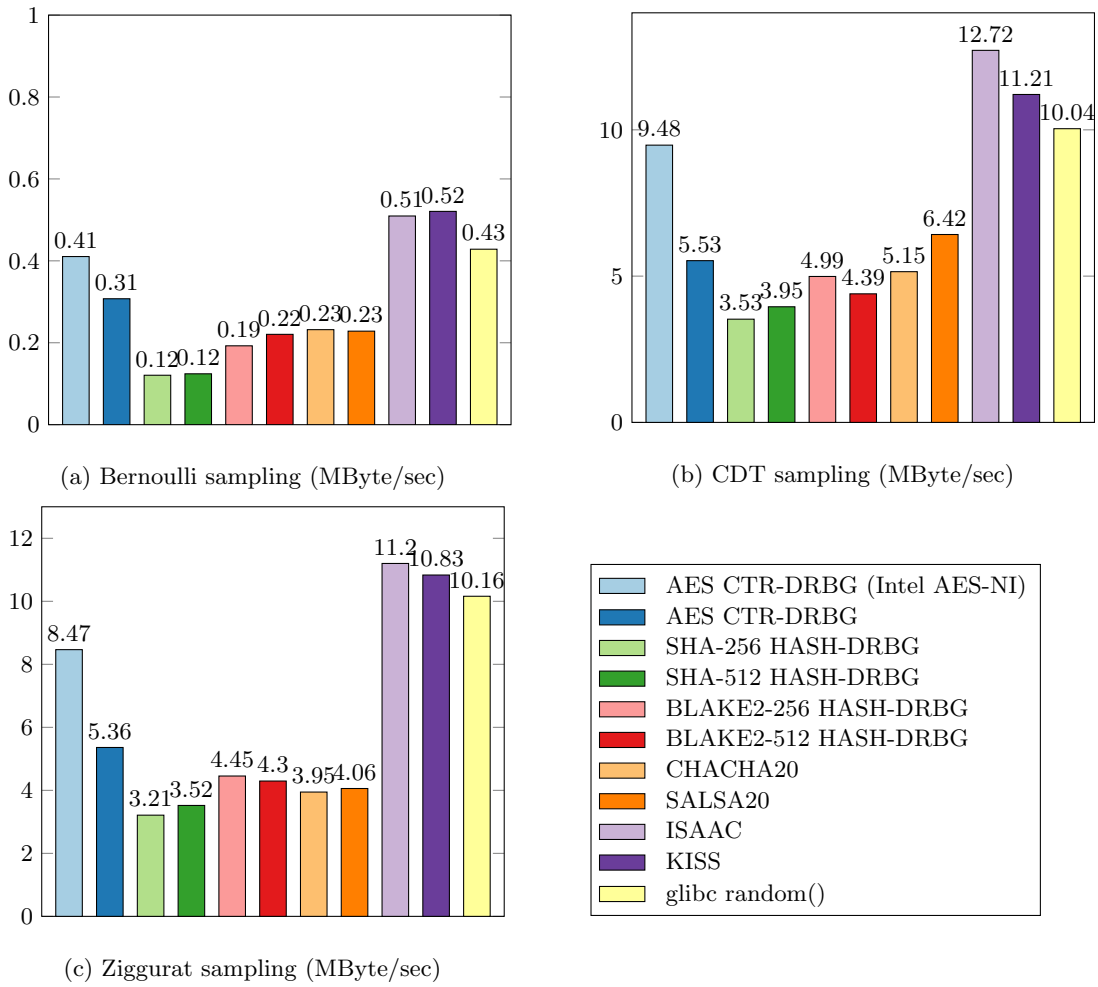


Fig. 4: Gaussian sampler performance comparison with various PRNG schemes [Intel i7 6700 CPU @ 3.4GHz] with GNU C Compiler auto-vectorisation *enabled*



(a) Bernoulli sampling (MByte/sec)

(b) CDT sampling (MByte/sec)

(c) Ziggurat sampling (MByte/sec)

Fig. 5: Gaussian sampler performance comparison with various PRNG schemes [Intel i7 6700 CPU @ 3.4GHz] with GNU C Compiler auto-vectorisation *disabled*

4.3 Memory Usage

Each of the PRNGs varies in its memory requirements as shown in Table 1. The large 64-bit arrays that must be maintained for every instance of ISAAC are the most demanding in terms of heap memory, while the SHA-2 implementation requires relatively significant heap and stack usage. The software implementation of AES is accelerated by making significant use of lookup tables that require 20kB of stack. Salsa20, ChaCha20 and the AES-NI PRNGs are the most efficient in terms of memory usage, illustrating the benefits of such stream ciphers and hardware acceleration when targetting a resource constrained platform.

Table 1: Memory usage of various PRNGs

PRNG	Heap (bytes) per instance	Stack (bytes)
glibc random()	0	0
KISS	48	0
ISAAC	4096	0
Salsa20	96	0
ChaCha20	96	0
SHA-256/512 HASH-DRBG	≥ 512	2240
AES CTR-DRBG (Intel AES-NI)	540	0
AES CTR-DRBG	540	20480

4.4 Consumption of Random Bytes

The efficiency of the Gaussian samplers in terms of their conversion of uniformly distributed random bytes to a Gaussian distribution is shown in Table 2. These figures were obtained using a testbench to generate 1M Gaussian samples and recording the number of random bytes required. CDT sampling directly translates a source of eight random bytes into a 64-bit table lookup operation. Ziggurat will utilize approximately 10% more random source bytes. Bernoulli is shown to be highly inefficient, requiring in excess of 32x the number of random samples required by CDT or Ziggurat.

Table 2: Consumption of random bytes by Gaussian Samplers

Gaussian Sampler	Number of Samples	Number of Random Bytes	Random Bytes / sample
Bernoulli	1048576	284292652	271.12
CDT	1048576	8388608	8
Ziggurat	1048576	9231120	8.8

5 Statistical Analysis

The following Section describes the statistical properties of the RNG's and their suitablility for use in Gaussian sampling and lattice-based schemes. We also show the effects of poor seeding of RNG's used within a Ziggurat Gaussian sampler, we have observed similar behaviour with the other samplers.

Within the sampling process, a good PRNG will not only generate random numbers, but should sample uniformly from the negative numbers equally as often as the positive. This is true of any equally

sized subsets of numbers over the sample range and it must apply to any sequential series of numbers generated by the PRNG. This is an ideal which will not be practically possible, but the amount by which an implementation differs should be within the confines of a random process. For example, we do not want the algorithm to produce alternately positive and negative numbers. We do, however, want a sample of numbers to be close to half positive, half negative.

For a distribution of N samples of $\{x_i | -t\sigma \leq x_i \leq t\sigma\}$, with tailcut factor t and $x_i \in \mathbb{Z}$, each of which is sampled n_i times and the mean value, μ, is calculated.

If one were to *begin with* a perfect distribution and continue to *sample from* a perfect distribution, the mean would begin at its exact value (the input parameter), but should be expected to vary sharply and rapidly thereafter. An important quality of the sampling process relating to the mean parameter is that the measured mean returns to values relatively close to it and, particularly, that it does so often. A measured mean value that stayed positive, as the number of samples increased, would result in the conclusion that the PRNG sampled too often from the positive set due to sign bit non-uniformities. Just as having a slowly varying mean would indicate erroneous behaviour, so too would be fluctuations between positive and negative values that rarely, if ever, return close to the required mean. For a discrete Gaussian sampler this behaviour can be seen in some implementations of the PRNG where it has not been seeded properly.

For samplers that are seeded correctly, Figure. 6(a) shows, sample by sample, how the measured mean (for mean parameter = 0) varies over a range of 1,000 samples, for the AES CTR-DRBG. Fig. 6(b) then shows the absolute value to clearly see how often it returns to zero. Fig. 6(c) and (d) show how the mean varies over a larger range of samples and over a range of σ, respectively. On a sample by sample basis we see rapidly varying profiles for the mean value. Figure 6(c) demonstrates the mean tending to zero as the number of samples increases, showing that the established mean is less influenced by additional samples, as N increases. In Fig. 6(d) we observe the opposite when σ is varied, i.e. the mean value grows, for a given number of samples, as σ is increased. This is due to the sampling of larger numbers in the equation of the mean.

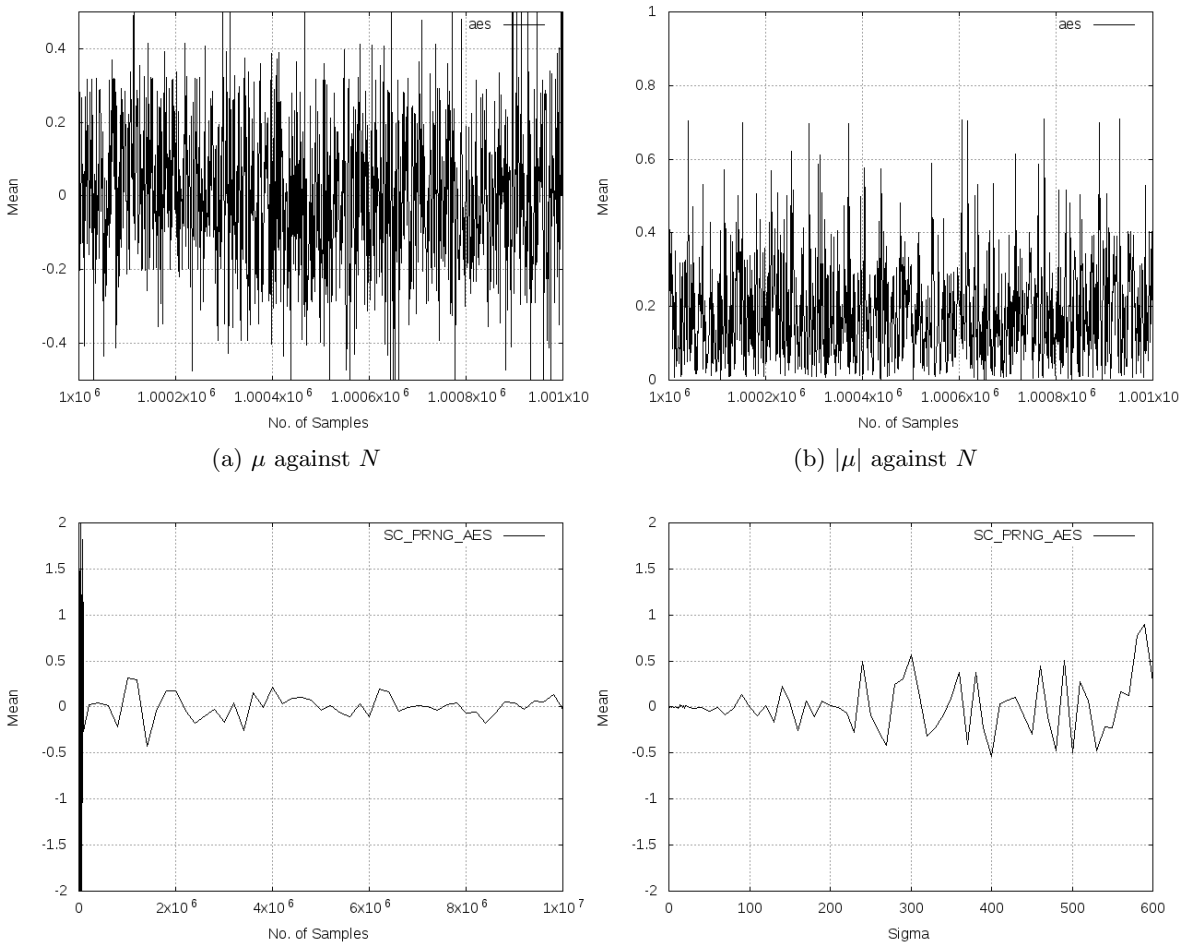
5.1 RNG Comparison

Here we show the results for a selection of the PRNGs with the Ziggurat sampler (similar results were observed for all samplers). In these experiments, when varying σ, the number of samples is $N = 1 \times 10^6$, and when varying the number of samples $N = 1 \times 10^7$ and $\sigma = 215$. Fig. 7 shows the sample-by-sample results for mean against number of samples. For these initial tests there is no indication of superiority amongst the PRNGs in relation to the statistical behaviour of the sampler, as seen in Figure 7, 8 and 9. We emphasise that these are initial statistical tests to build a level of confidence in the RNGs and Gaussian samplers, prior to performing tests using established statistical suites.

5.2 Seeding the PRNGs

The criteria for CSPRNGs is more complex than those of PRNGs. An important consideration among CSPRNGs is robustness against improper seeding. Seeds for PRNGs should have a sufficient information density to allow the period and other properties to reach their maximum. Fig. 10 and Fig. 11 show the effects of removing the seeding control. In our test environment, we are able to control the entropy source of seeding such that any predetermined values can be used. In order to remove the seeding control we simply replace a good entropy source with a sequence of zero bytes.

In most of the samplers, by removing this seeding control we observe the introduction of a bias within the Gaussian sampler. Figures 10 to 12 demonstrate the occurence of such a bias. The exceptions to this are AES and ChaCha20/Salsa20. These CSPRNGs show a clear robustness against improper seeding. The HASH-DRBG results concern us as we have validated the Hash-DRBG functionality using NIST test vectors, but we expected the backtracking resistance to overcome the improper seeding issues. This is likely to be an implementation error. We have not investigated this behaviour further as the performance of the HASH-DRBG is inferior to most other PRNG's under investigation, however we will thoroughly review our HASH-DRBG implementation in order to clarify the source of the error.



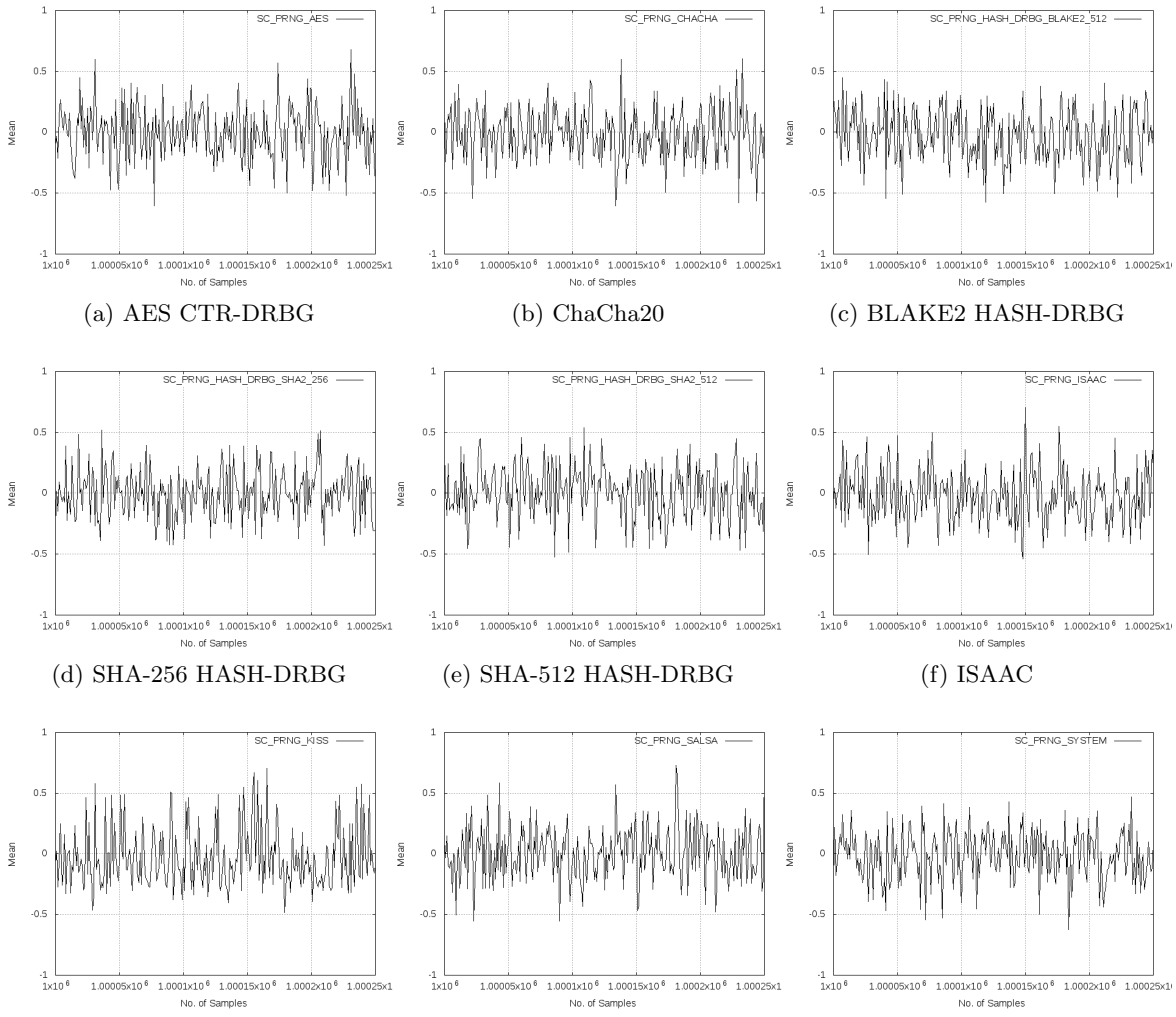
(a) μ against N

(b) $|\mu|$ against N

(c) μ against N

(d) μ against σ parameter

Fig. 6: Plots of (a), the mean, and (b), the absolute value of the mean, against the number of samples for the Ziggurat sampler at $\sigma = 215$ with the AES PRNG. It can be seen in (a) that the mean does not stay positive or negative over substantial consecutive numbers of samples. From (b) it can be seen that the mean regularly returns to values close to the parameter. Both plots show sharp, rapid variations. Plots (c) and (d) show how the mean varies over a large range of samples and over the standard deviation parameter, respectively. Here, the graphs should be decreasing and increasing, respectively. Notice the thick lines at low number of samples in (d), which show large deviations until a mean is established.

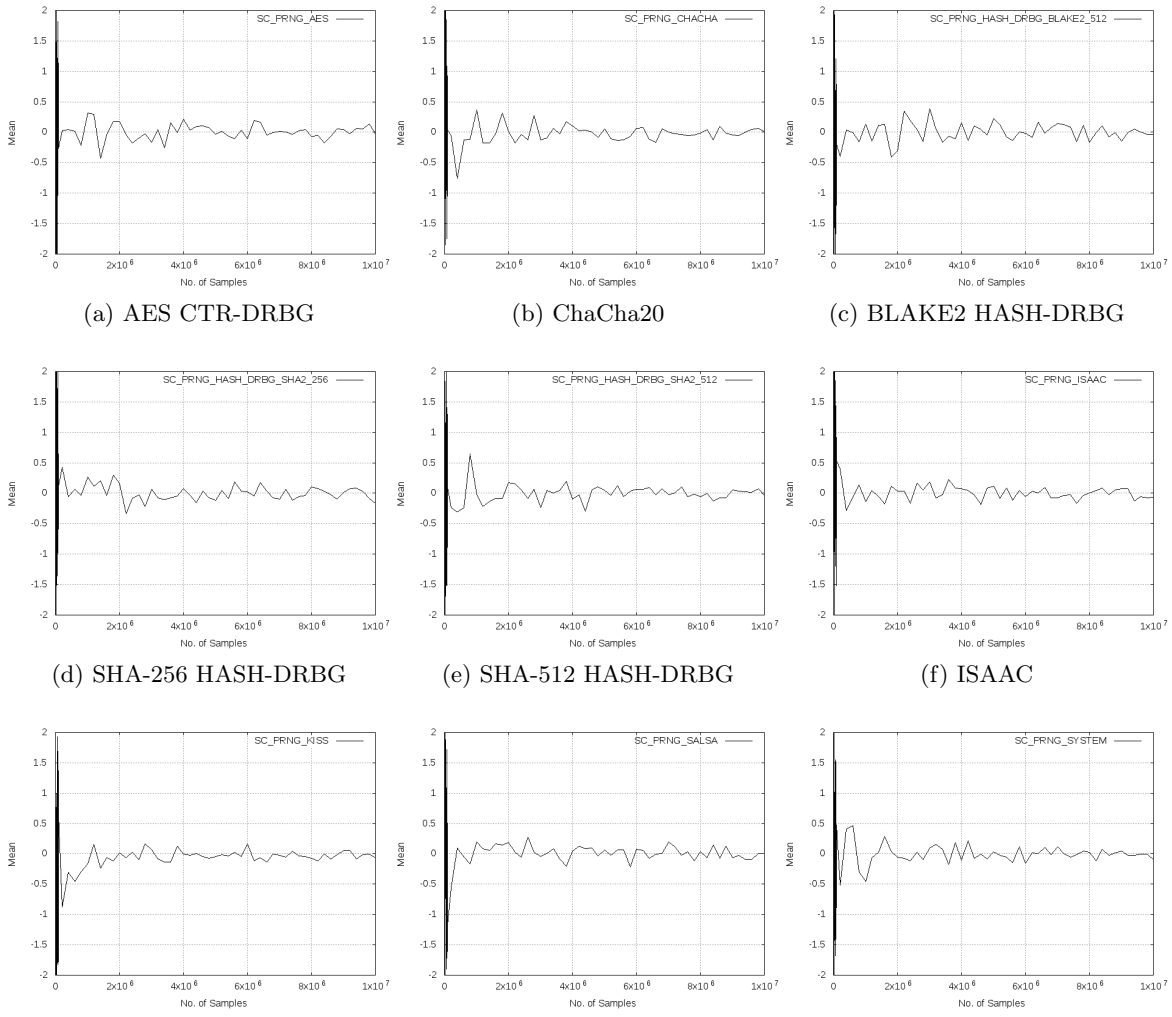


(a) AES CTR-DRBG (b) ChaCha20 (c) BLAKE2 HASH-DRBG

(d) SHA-256 HASH-DRBG (e) SHA-512 HASH-DRBG (f) ISAAC

(g) KISS (h) Salsa20 (i) glibc random()

Fig. 7: Mean against the number of samples, starting from 1 million, for 250 samples. A selection of the PRNGs are shown, but graphs are representative of the whole set analysed. The figure shows expected behaviour across the range of PRNGs.

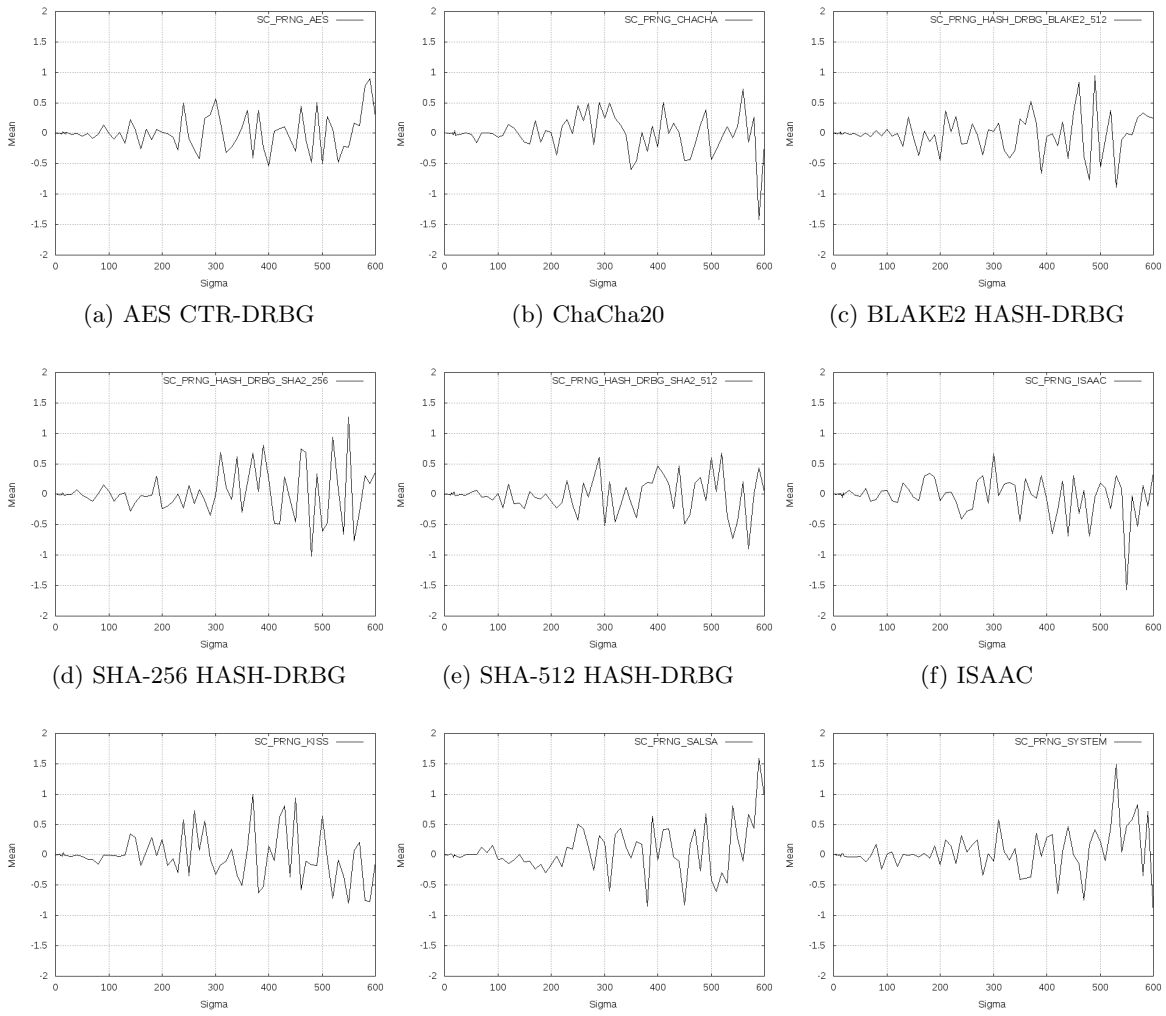


(a) AES CTR-DRBG (b) ChaCha20 (c) BLAKE2 HASH-DRBG

(d) SHA-256 HASH-DRBG (e) SHA-512 HASH-DRBG (f) ISAAC

(g) KISS (h) Salsa20 (i) glibc random()

Fig. 8: Mean against number of samples over a larger range of samples. The combination of tending to zero in this figure and the rapid variations of the last figure, provide evidence of the samplers behaving statistically well, regardless of the PRNG used.

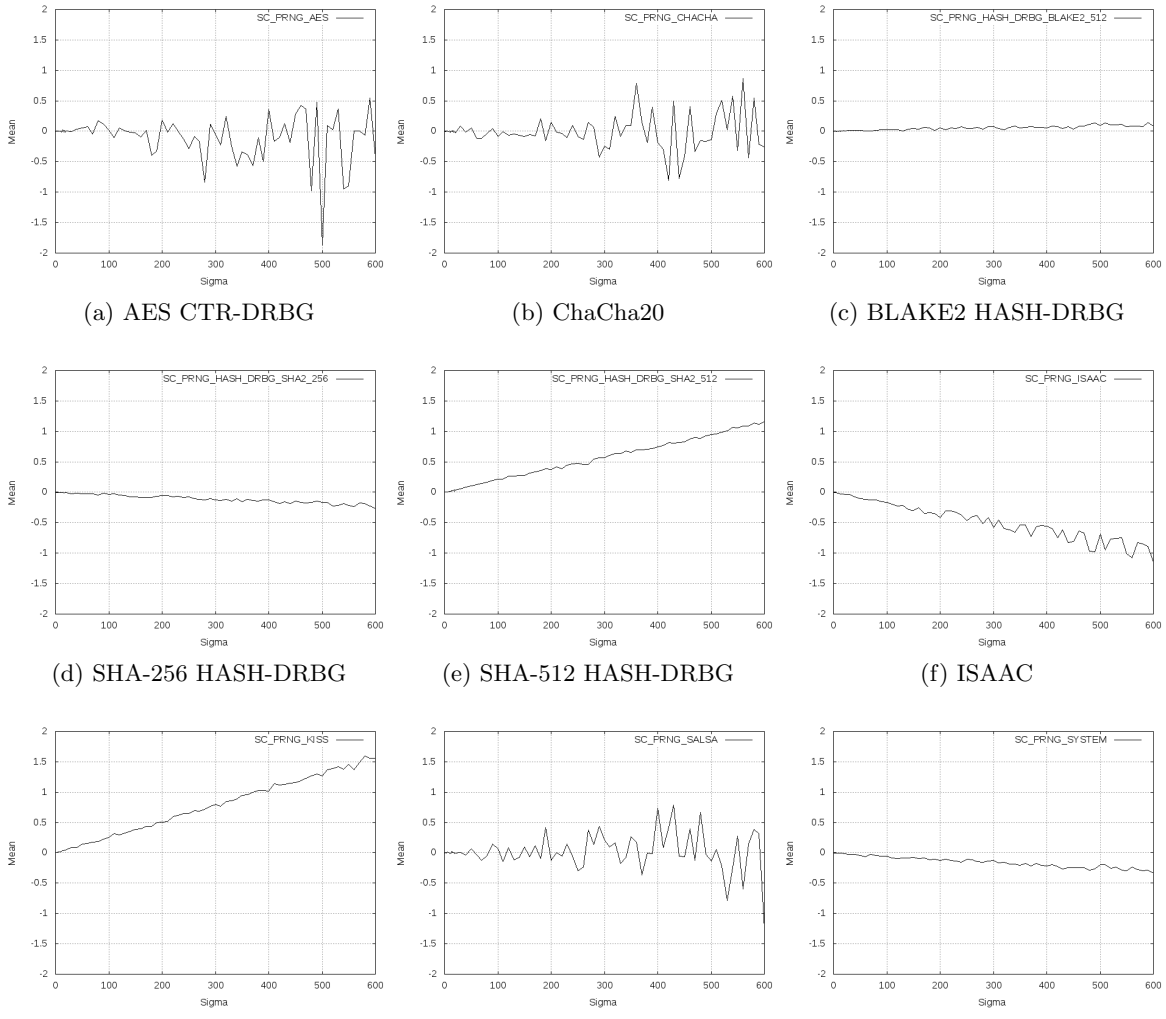


(a) AES CTR-DRBG (b) ChaCha20 (c) BLAKE2 HASH-DRBG

(d) SHA-256 HASH-DRBG (e) SHA-512 HASH-DRBG (f) ISAAC

(g) KISS (h) Salsa20 (i) glibc random()

Fig. 9: Mean against *sigma* for the various PRNGs. The figure provides further evidence that the sampler works well with any of the PRNGs.

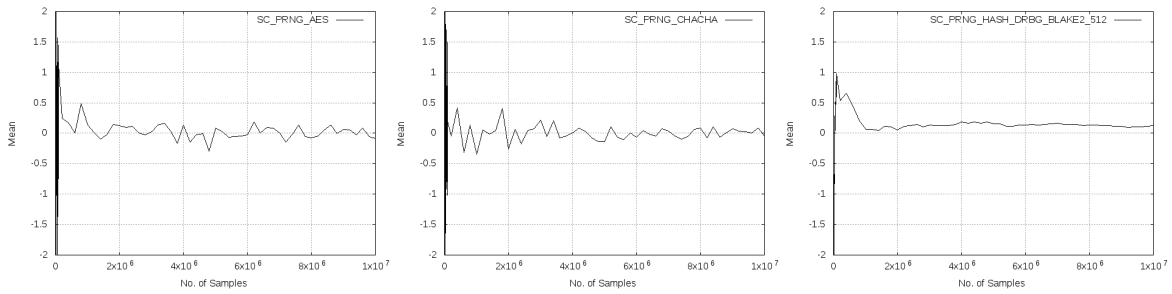


(a) AES CTR-DRBG (b) ChaCha20 (c) BLAKE2 HASH-DRBG

(d) SHA-256 HASH-DRBG (e) SHA-512 HASH-DRBG (f) ISAAC

(g) KISS (h) Salsa20 (i) glibc random()

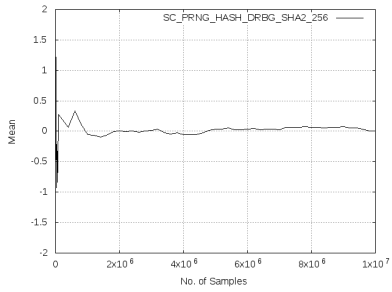
Fig. 10: Mean against *sigma* for the various PRNGs with seeding control removed. The figure shows biases in the mean value for all but AES, CHACHA and SALSA.



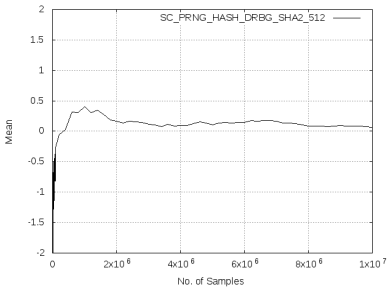
(a) AES CTR-DRBG

(b) ChaCha20

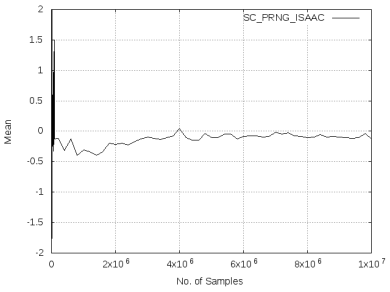
(c) BLAKE2 HASH-DRBG



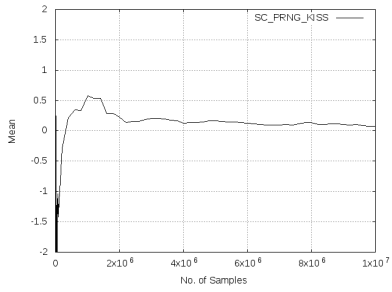
(d) SHA-256 HASH-DRBG



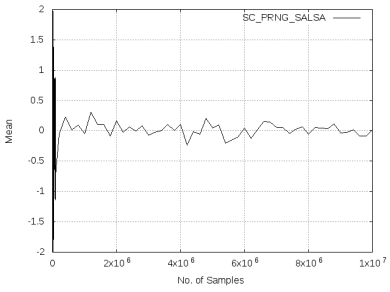
(e) SHA-512 HASH-DRBG



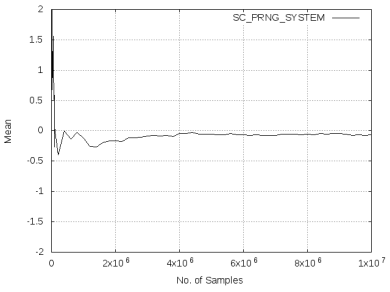
(f) ISAAC



(g) KISS

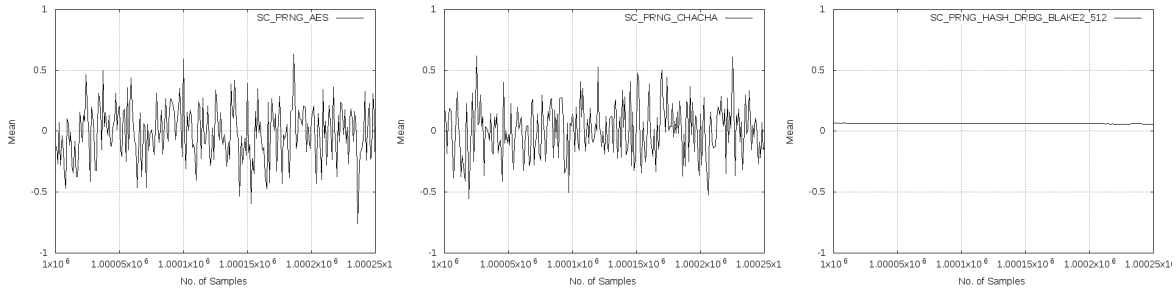


(h) Salsa20



(i) glibc random()

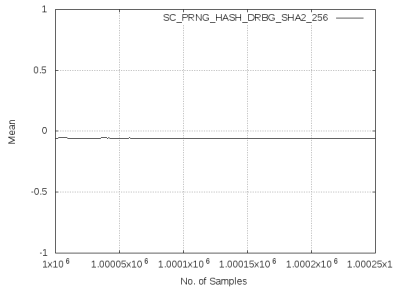
Fig. 11: Mean against number of samples for the various PRNGs with seeing control removed. The figure shows biases in the mean value for all but AES, CHACHA and SALSA.



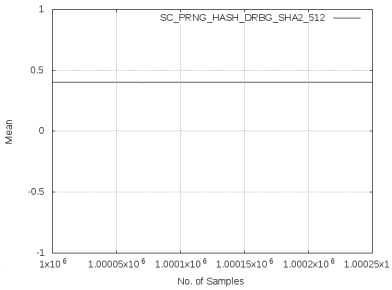
(a) AES CTR-DRBG

(b) ChaCha20

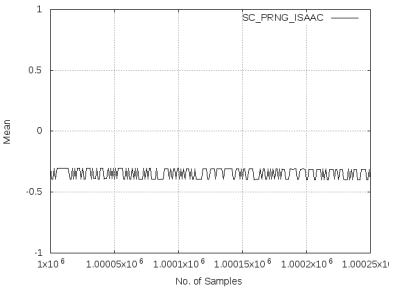
(c) BLAKE2 HASH-DRBG



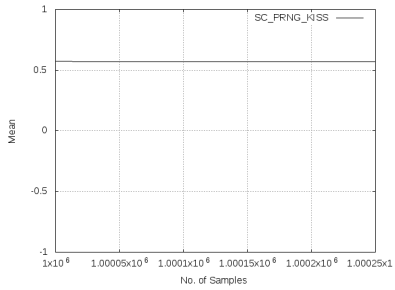
(d) SHA-256 HASH-DRBG



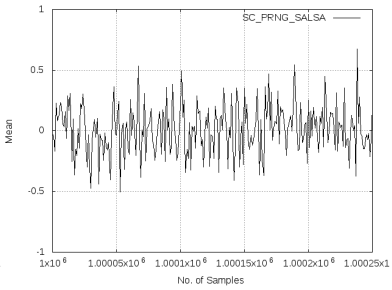
(e) SHA-512 HASH-DRBG



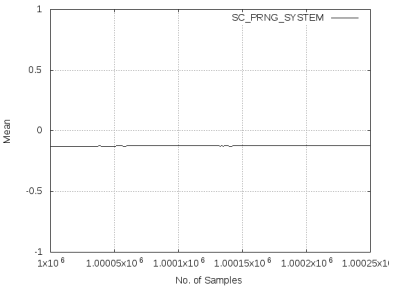
(f) ISAAC



(g) KISS



(h) Salsa20



(i) glibc random()

Fig. 12: Sample-by-sample results for poor seeding. We see that the majority of samplers introduce bias when the seeding is not controlled.

6 Conclusion

For generic implementations of discrete Gaussian sampling CTR-DRBG with AES is a safe option, particularly if AES hardware acceleration is available or you want to enter a NIST competition. It possesses a balance of performance and security, is well understood and is accepted by the security community. HASH-DRBG did not perform well in our tests, however this does not exclude hashes from our future investigations. ISAAC and KISS are a very good choice for high performance testing, but their lack of suitable cryptographic credentials will confine their use to testing only where they offer superior performance to glibc random(). The stream ciphers ChaCha20 and Salsa20 offer reasonable performance and are of particular interest for constrained devices due to their low memory utilisation. System entropy stores will be used only for the purpose of initial seeding. We have conducted extensive testing of core components and various schemes within Lattice-based cryptography as part of the SAFEcrypto project. We will be releasing open-source implementations over the course of the NIST post-quantum submission period.

7 Acknowledgements

The SAFEcrypto project has received funding from the European Union Horizon 2020 research and innovation program under grant agreement No. 644729.

References

1. National Institute of Standards and Technology, "Post-Quantum Crypto Standardization, Call For Proposals Announcement," 2017.
2. L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky, "Lattice signatures and bimodal Gaussians," *IACR Cryptology ePrint Archive*, vol. 2013, p. 383, 2013. Full version of [31].
3. T. Pöppelmann, L. Ducas, and T. Güneysu, "Enhanced Lattice-Based Signatures on Reconfigurable Hardware," in *Cryptographic Hardware and Embedded Systems CHES 2014*, CHES '14, (Busan, South Korea), pp. 353–370, Springer Berlin Heidelberg, 2014.
4. D. Knuth and A. Yao, *Algorithms and Complexity: New Directions and Recent Results*, ch. The Complexity Of Nonuniform Random Number Generation. Academic Press, 1976.
5. C. Peikert, "An efficient and parallel Gaussian sampler for lattices," in *International Cryptoology Conference CRYPTO 2010*, CRYPTO '10, (Santa Barbara, CA, USA), Springer Berlin Heidelberg, 2010.
6. G. Marsaglia and W. W. Tsang, "The Ziggurat Method for Generating Random Variables," *Journal of Statistical Software*, vol. 5, no. 1, pp. 1–7, 2000.
7. J. A. Doornik, *An Improved Ziggurat Method to Generate Normal Random Samples*. University of Oxford.
8. M.-J. O. Saarinen, "Gaussian Sampling Precision in Lattice Cryptography," Tech. Rep. 953, 2015.
9. J. Howe, C. Moore, M. O'Neill, F. Regazzoni, T. Güneysu, and K. Beeden, "Lattice-based Encryption Over Standard Lattices In Hardware," in *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, (Austin, TX, USA), ACM, 2016.
10. T. Oder, T. Pöppelmann, and T. Güneysu, "Beyond ECDSA and RSA: lattice-based digital signatures on constrained devices," in *DAC '14*, pp. 1–6, ACM, 2014.
11. M. O. Saarinen, "Hilabliss." `https://github.com/mjosaarinen/hilabliss`, 2016.
12. D. J. Bernstein, "ChaCha, a variant of Salsa20," tech. rep., ECRYPT.
13. E. Alkim, L. Ducas, T. Pppelmann, and P. Schwabe, "Post-quantum key exchange - a new hope." Cryptology ePrint Archive, Report 2015/1092, `http://eprint.iacr.org/2015/1092`, 2015. Accessed: 2017-03-27.
14. A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson, "ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)," RFC 7905, RFC Editor, June 2016.
15. E. Barker and J. Kelsey, "NIST Special Publication 800-90A Revision 1," tech. rep., National Institute of Standards and Technology, June 2015.
16. B. Gladman, "AES C Source Code." `https://github.com/BrianGladman/AES`, 2006. Accessed: 2017-03-27.
17. L. Ducas, "Accelerating bliss: the geometry of ternary polynomials," *IACR Cryptology ePrint Archive*, vol. 2014, p. 874, 2014.

18. L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky, "Bliss: Bimodal lattice signature schemes." `http://bliss.di.ens.fr`, 2013.
19. B. Gladman, "SHA1, SHA2, HMAC and Key Derivation in C." `http://www.gladman.me.uk/`, 2002. Accessed: 2017-03-27.
20. Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols," RFC 7539, RFC Editor, May 2015.
21. D. J. Bernstein, "The Salsa20 core." `https://cr.yp.to/snuffle.html`, 2007. Accessed: 2017-03-27.
22. D. J. Bernstein, "The ChaCha family of stream ciphers." `https://cr.yp.to/chacha.html`, 2008. Accessed: 2017-03-27.
23. Y. Nir, "ChaCha20, Poly1305, and Their Use in the Internet Key Exchange Protocol (IKE) and IPsec," RFC 7634, RFC Editor, August 2015.
24. R. J. J. Jr., "Fast Software Encryption," in *Proceedings of the Fast Software Encryption Third International Workshop*, FSE 1996, (Cambridge, UK), pp. 41–49, Springer Berlin Heidelberg, 1996.
25. B. Jenkins, "ISAAC: a fast cryptographic random number generator." `http://burtleburtle.net/bob/rand/isaacafa.html`, 1996. Accessed: 2017-03-27.
26. G. Marsaglia and A. zaman, "The kiss generator," tech. rep., Technical report, Department of Statistics, Florida State University, Tallahassee, FL, USA, 1993.
27. G. Rose and Q. Incoroporated, "KISS: A Bit Too Simple,"
28. IEEE, *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992, Open Group Technical Standard Base Specifications, Issue 6.
29. Software Engineering Institute, "SEI CERT C Coding Standard," coding standard, Carnegie Mellon University, 2016.
30. D. A. O. Joppe W. Bos and D. Stefan, "Fast Implementations of AES on Various Platforms," *IACR Cryptology ePrint Archive*, vol. 2009, p. 501, 2009.
31. L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky, "Lattice signatures and bimodal Gaussians," in *CRYPTO (1)* (R. Canetti and J. A. Garay, eds.), vol. 8042 of *LNCS*, pp. 40–56, Springer, 2013. Proceedings version of [2].