Pseudo-random Number Generation on the IBM 4758 Secure Crypto Coprocessor

Nick Howgrave-Graham, Joan Dyer, and Rosario Gennaro

IBM T.J.Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598.
nahg,joandy,rosario@watson.ibm.com

Abstract. In this paper we explore pseudo-random number generation on the IBM 4758 Secure Crypto Coprocessor. In particular we compare several variants of Gennaro's provably secure generator, proposed at Crypto 2000, with more standard techniques based on the SHA-1 compression function. Our results show how the presence of hardware support for modular multiplication and exponentiation affects these algorithms.

1 Introduction

The use of cryptographic techniques is a key element of modern e-business applications. Such applications use cryptography in a variety of ways to protect the privacy and confidentiality of data, to ensure the integrity of data, and to provide user accountability through digital signature techniques.

The security of cryptographic algorithms in real life applications, however relies mostly on two main assumptions:

1. that the secret keys used in the algorithms have not been compromised,
2. that the code executing the algorithms is really performing the tasks that it is supposed to.

Thus, in real life there is a concrete need to address these issues: the physical security of the keys and the code used in cryptographic algorithms. This is why most of the time, the keys are stored in a secure, protected memory device which is not easily tampered with. Similarly the code must be run in a protected environment. One answer to these issues is to use a *secure coprocessor*.

A secure coprocessor is a device that offloads computationally intensive cryptographic processes from the hosting server, and performs sensitive tasks unsuitable for less secure general purpose computers. Depending on the applications, it may be a special-purpose computational engine (say a hardware RSA chip), or it may be more useful to have a general-purpose computing environment. Such a device must withstand physical and logical attacks; it must run the programs that it is supposed to, unmolested. The host server should be able to (remotely) distinguish between the real device and a possible impersonation. The coprocessor must remain secure even if adversaries carry out destructive analysis of one or more devices.

Ç.K. Koç, D. Naccache, and C. Paar (Eds.): CHES 2001, LNCS 2162, pp. 93–102, 2001.

An important class of secure coprocessors are the so-called *field programmable* ones, which allow the user to write custom software for the device, which then loads it under some controlled condition and subsequently runs it.

In this paper we consider the IBM 4758 PCI Secure Crypto Coprocessor, which is an example of such a field programmable device [12]. The IBM 4758 is the only programmable device on the market which has been certified at FIPS 140-1 Level 4, the highest security classification for a commercial cryptographic device [8]. We elaborate more on the technical specifications of the IBM 4758 in Section 2.

We report the implementation results, on the IBM 4758, of a random number generator recently proposed at CRYPTO'2000 [5].

1.1 The Problem of Pseudo-random Bit Generation

Many, if not all, cryptographic algorithms rely on the availability of truly random bits. However perfect randomness is a scarce resource. Fortunately for almost all cryptographic applications, it is sufficient to use pseudo-random bits, i.e. sources of randomness that "look" sufficiently random to the adversary.

This notion can be made more formal. The concept of cryptographically strong pseudo-random bit generators (PRBG) was introduced in papers by Blum and Micali [3] and Yao [14]. Informally a PRBG is cryptographically strong if it passes all polynomial-time statistical tests or, in other words, if the distribution of sequences output by the generator cannot be distinguished from truly random sequences by any polynomial-time judge.

A PRBG is called *provably secure*, if its security can be reduced to a well-established conjectured hard problem (like factoring or computing discrete logarithms.)

[5] assumes a variation of the Discrete Log Assumption. More specifically it assumes that if solving the discrete log problem modulo an n-bit prime p is hard even when the exponent is small (say only c bits long with $c < n$), then the function $f : \{0,1\}^c \longrightarrow Z_p^*$ defined as $f(x) = g^x \bmod p$ has strong pseudo-randomness properties over Z_p^*. In particular it is possible to think of it as a pseudo-random generator itself. By iterating the above function and outputting the appropriate bits, an efficient pseudo-random bit generator is obtained. The generator outputs $n - c - 1$ bits per iteration, which consists of a single exponentiation with a c-bit exponent.

An attractive feature of this generator is that all the exponentiations are computed over a fixed basis, and thus precomputation tables can be used to speed them up.

Using typical parameters $n = 1024$ and $c = 160$ we obtain roughly 860 pseudo-random bits per 160-bit exponent exponentiations. Using the precomputation scheme proposed in [7] one can show that such exponentiation will cost on average roughly 40 multiplications, using a table of only 12 Kbytes. Thus we obtain a rate of more than 21 pseudo-random bits per modular multiplication. Different tradeoffs between memory and efficiency can be obtained.

1.2 Interesting Questions and Our Results

When we started this implementation project we had the following questions which we thought were worth investigating:

- The IBM 4758, as many other crypto coprocessors, provides hardware support for modular math operations (modular multiplications and exponentiations). How effective are precomputation techniques like [7] in the presence of hardware support? Is the extra storage worth the potential gain in speed?
- The generator proposed in [5] is the fastest provably secure PRBG in the literature, based on established number theoretic conjectures. It would be interesting to know how it compares to other PRBGs whose security is assumed "from scratch" since they are related to block ciphers and hash functions. In particular it is interesting to see the results of this comparison in a constrained computing environment like a secure coprocessor.

For the first question, we ran the algorithm with various settings of the [7] precomputation scheme, as well as with no precomputation at all. In the latter case, modular exponentiations were computed completely in hardware, while in the former case the dedicated hardware was invoked only for modular multiplications. Quite surprisingly we obtained timing results that showed no increase in speed with the use of precomputation tables. Actually the algorithm was substantially slowed down. This seems to indicate that hardware support for modular exponentiations totally eliminates the need for precomputation schemes.

For the second question, we ran the [5] generator against an implementation of a pseudo-random generator consistent with the ANSI X9.17 Key Management standard[1]. This generator is based on repeated application of the hash function SHA-1. The timing results show that it is still considerably more efficient than our number theoretic construction (but, as mentioned above, this is at the cost of not being able to be proven to be reducible to any (supposed) hard mathematical problem).

2 The IBM 4758 Architecture

The IBM 4758 Secure Crypto Coprocessor is a hardware card, that plugs into industry-standard PCI slots in personal computers and other systems that support the PCI bus. The Coprocessor secure processing environment contains a 486-compatible microcoprocessor, custom hardware to perform DES and public key cryptographic algorithms, a secure clock/calendar, and a hardware random number generator. See Figure 1 for a complete list of specifications.

It also has protective shields, sensors, and control circuitry to protect against a wide variety of attacks against the system. More specifically the 4758 is protected against attacks involving probe penetration, power sequencing, radiation

[1] In fact it is the implementation that is used by the card itself for pseudo-random number generation.

Features:	*Card type:*	PCI 32-bit Bus Master
	Internal Processor:	486 99MHz
	RAM:	4 Mbytes
	ROM/FLASH:	4 Mbytes
	Battery-backed RAM:	32 Kbytes
Crypto:	*DES:*	Hardware support
	RSA, DSS:	Software with hardware support for 1024-bit modular math
	Hashing:	SHA-1 in hardware
	Random numbers:	Noise-based hardware RNG

Fig. 1. Features of the IBM 4758

and temperature manipulation, consistent with the FIPS 140-1 Level 4 Certification. The basic element of the protective layer is a grid of conductors which is monitored by circuitry that can detect changes in the properties of the conductors. The conductors themselves are non-metallic and closely resemble the material they are embedded in. This makes discovery, isolation and manipulation all the more difficult. These grids are arranged in several layers and the sensing circuitry can detect accidental connections between layers as well as changes in an individual layer. The sensing grids are made of flexible material and are wrapped around and attached to the secure processor package as if it were being gift-wrapped. After the package is wrapped, it is embedded in a potting material (which as mentioned closely resembles the conductors). Finally the entire package is enclosed in a grounded shield to reduce susceptibility to electromagnetic interference and to reduce detectable electromagnetic emanations.

During the final manufacturing step, the Coprocessor generates a unique public key pair, which is stored in the device. The tamper detection circuitry is activated at this time and remains active throughout the useful life of the Coprocessor, protecting this private key, as well as all other keys and sensitive data. The Coprocessor public key is certified at the factory by a global IBM private key and the certificate is retained in the Coprocessor. Subsequently, the Coprocessor private key is used to sign the Coprocessor status responses which in conjunction with the public key certificate, demonstrate that the Coprocessor remains intact and is genuine.

From the time of manufacture, if the tamper sensors are ever triggered, the Coprocessor zeroizes its critical keys, destroys its certification, and is rendered inoperable.

2.1 Developing Applications for the 4758

The Coprocessor contains firmware to manage its specialized hardware and to control loading of additional software. The card runs the IBM CP/Q embedded operating system, which has been extended with device drivers and other features specific to the Coprocessor. The resulting control program, CP/Q^{++}, provides the platform for application development. A complete custom application (like our pseudorandom generator) can be built on the CP/Q^{++} environment.

During development, the security features of the 4758 and the public key signatures used to validate download requests are irrelevant, but enabling symbolic debugging capability by adding a debug probe to CP/Q^{++} is essential. Preparing the 4758 for development is a one-time process. This step allows an external party to identify a 4758 by means of the identification of the officer assigned to the operating system layer. It is important in the overall picture of security provided by a 4758, in that a card with debug capability cannot masquerade as a secure card to the external world.

Application code is written in C, with one portion destined for the 4758 and the other its partner on the host machine. The 4758-based software is cross-compiled using supplied headers. After the normal link step, there are a few additional steps:

1. translation from host-native executable format to the format accepted by the CP/Q^{++} loader, as well as translating debug symbols to a format understood by the symbolic debugger supplied with the Toolkit
2. packing the translated executable into a disk image for the read-only file system within the 4758, used by CP/Q^{++}
3. downloading the disk image

The last, download, step is a bit lengthy in that the 4758 must be rebooted in order to open the hardware locks that protect flash to enable writing of code, and the hardware is tested each time the 4758 is reset (essential parts of the security architecture).

After development has completed, software can be deployed using any of the host platforms for which a device driver is available (includes AIX, OS/2, Linux, others). The development Toolkit is available for NT, with a version hosted in Linux to appear shortly.

3 The New Pseudorandom Generator

In this section we briefly recall the [5] generator.

Number-Theoretic Preliminaries. Let p be a prime. We denote with n the binary length of p. It is well known that $Z_p^* = \{x : 1 \leq x \leq p-1\}$ is a cyclic group under multiplication $\bmod p$. Let g be a generator of Z_p^*. Thus the function

$$f : Z_{p-1} \longrightarrow Z_p^*$$

$$f(x) = g^x \bmod p$$

is a permutation. The inverse of f (called the *discrete logarithm* function) is conjectured to be a function hard to compute (the cryptographic relevance of this conjecture first appears in the seminal paper by Diffie and Hellman [4] on public-key cryptography). The best known algorithm to compute discrete logarithms is the so-called *index calculus* method [1] which however runs in time sub-exponential in n.

In some applications (like the one we are going to describe in this paper) it is important to speed up the computation of the function $f(x) = g^x$. One possible way to do this is to restrict its input to small values of x. Let c be a integer which we can think as depending on n ($c = c(n)$). Assume now that we are given $y = g^x \bmod p$ with $x \leq 2^c$. It appears to be reasonable to assume that computing the discrete logarithm of y is still hard even if we know that $x \leq 2^c$. Indeed the running time of the index-calculus method depends only on the size n of the whole group. Depending on the size of c, different methods may actually be more efficient. Indeed the so-called *baby-step giant-step* algorithm by Shanks [6] or the *rho* and *lambda* algorithms by Pollard [10] can compute the discrete log of y in $O(2^{c/2})$ time. If one restricts the field to *generic* algorithms (i.e. algorithms that can only perform group operations and cannot take advantage of specific properties of the encoding of group elements) then Schnorr in [11] proves that this is the best that can be done.

If the complete factorization of $p-1$ is known, then the running time of these algorithms can be improved by using the Pohlig-Hellman decomposition [9]. This is done by reducing the original discrete log problem, into several "smaller" problems (one for each distinct prime factor in $p-1$).

Van Oorschot and Wiener in [13] present a new method of combining the Pollard *lambda* method with a partial Pohlig-Hellman decomposition. Their end result is that for *random* primes, using short exponents is *not* secure. However their attack can be avoided by restricting the moduli to be *safe* primes p (i.e. such that $\frac{p-1}{2}$ is also a prime) since in this case the Polhig-Hellman decomposition is useless.

Thus if we set $c = \omega(\log n)$, there are no known polynomial time algorithms that can compute the discrete log of $y = g^x \bmod p$ when $x \leq 2^c$ and p is a safe prime. One can explicitly assumed that *no* such efficient algorithm can exist. This is called the *Discrete Logarithm with Short c-Bit Exponents (c-DLSE)* Assumption and we will adopt it as the basis of our results as well.

Assumption 1 (c-DLSE) *Let $SPRIMES(n)$ be the set of n-bit safe primes and let c be a quantity that grows faster than $\log n$ (i.e. $c = \omega(\log n)$). For every probabilistic polynomial time Turing machine $\mathcal{I}$, for every polynomial $P(\cdot)$ and for sufficiently large n we have that*

$$\Pr\begin{bmatrix} p \leftarrow SPRIMES(n); \\ x \leftarrow R_c; \\ \mathcal{I}(p, g, g^x, c) = x \end{bmatrix} \leq \frac{1}{P(n)}$$

In practice, given today's computing power and discrete-log computing algorithms, it seems to be sufficient to set $n = 1024$ and $c = 160$. This implies a "security level" of 2^{80} (intended as work needed in order to "break" 160-DLSE).

3.1 The Algorithm

Consider the following function:

$$\mathsf{RG}_{n,c} : Z_{p-1} \longrightarrow Z_p^* \qquad \mathsf{RG}_{n,c}(s) = \hat{g}^{(s \text{ div } 2^{n-c})} g^{s_1} \bmod p$$

That is we consider modular exponentiation in Z_p^* with base g, but only after zeroing the bits in positions $2, \ldots, n-c$ of the input s (these bits are basically ignored).

The function RG induces a distribution over Z_p^* in the usual way. We denote it with $RG_{n,c}$ the following probability distribution over Z_p^*

$$Prob_{RG_{n,c}}[y] = Prob[y = \mathsf{RG}_{n,c}(s) \ ; \ s \leftarrow Z_{p-1}]$$

It is possible to prove (see [5]) that the distribution $RG_{n,c}$ is computationally indistinguishable from the uniform distribution over Z_p^* if the c-DLSE assumption holds.

It is now straightforward to construct the new generator. The algorithm receives as a seed a random element s in Z_{p-1} and then it iterates the function RG on it. The pseudo-random bits outputted by the generator are the bits ignored by the function RG. The output of the function RG will serve as the new input for the next iteration.

In more detail, the algorithm $\mathsf{IRG}_{n,c}$ (for Iterated-RG generator) works as follows. Start with $x^{(0)} \in_R Z_{p-1}$. Set $x^{(i)} = \mathsf{RG}_{n,c}(x^{(i-1)})$. Set also $r^{(i)} = x_2^{(i)}, x_3^{(i)}, \ldots, x_{n-c}^{(i)}$. The output of the generator will be $r^{(0)}, r^{(1)}, \ldots, r^{(k)}$ where k is the number of iterations (chosen such that $k = poly(n)$ and $k(n-c-1) > n$).

Notice that this generator outputs $n-c-1$ pseudo-random bits at the cost of a modular exponentiation with a random c-bit exponent (i.e. the cost of the computation of the function RG).

3.2 Efficiency Analysis

Let's fix $n = 1024$ and $c = 160$. With these parameters we can safely assume that the complexity of the best known algorithms to break c-DLSE is beyond the reach of today's computing capabilities.

We obtain 863 bits at the cost of roughly 240 multiplications, which yields a rate of about 3.5 bits per modular multiplication. The most expensive part of the computation of our generator is to compute $\hat{g}^s \bmod p$ where s is a c-bit value.

We can take advantage of the fact that the modular exponentiations are all computed over the same basis $\hat{g}$. This feature allows us to precompute powers of $\hat{g}$ and store them in a table, and then use this values to compute fastly $\hat{g}^s$ for any s.

Lim and Lee [7] present flexible trade-offs between memory and computation time to compute exponentiations over a fixed basis. Their approach is applicable to our scheme as well. In short, the [7] precomputation scheme is governed by two parameters h, v. The storage requirement is $(2^h - 1)v$ elements of the field. The number of multiplications required to exponentiate to a c-bit exponent is $\left\lceil \frac{c}{h} \right\rceil + \left\lceil \frac{c}{hv} \right\rceil - 2$ in the worst case.

Using the choice of parameters for 160-bit exponents suggested in [7] we can get roughly 40 multiplications with a table of only 12 Kbytes. This yields a rate of more than 21 pseudo-random bits per multiplication. A large memory

implementation (300 Kbytes) will yield a rate of roughly 43 pseudo-random bits per multiplication.

4 Implementation Timing Results

We ran a C implementation of the above generator on the IBM 4758 card using the implementation procedures described in Section 2.1. In particular this means that we used the 4758 native hardware support for modular exponentiations and modular multiplications.

We first ran 1024 iterations of the generator (i.e. an output of 863 Kbits) without using precomputation tables. The task took approximately 4.75 seconds, which implies a rate of 22.7 Kbytes/sec. Thus, for example, this is the rate at which two secure coprocessors can securely encrypt data (via symmetric encryption) under a strong mathematical guarantee of security.

We then ran the algorithm using the [7] precomputation scheme with various settings of the parameters h, v described above. The experimental results confirmed the theoretical speed-ups between different choices of the parameters, however they also demonstrated a major slowdown of the algorithm compared to the case in which we computed the whole exponentiation in hardware.

The explanation is that the overhead of invoking in software the hardware chip for modular multiplication several times, offset whatever gain we could obtain in decreasing the number of multiplications by use of precomputation tables.

The results are summarized in Figure 2.

(h, v)	Storage (Kbytes)	Time (sec)
–	0	4.75
(5,8)	32	79.33
(8,2)	64	68.4
(8,4)	128	57.21
(8,5)	160	55.10
(8,10)	320	50.82
(10,4)	512	46.41
(10,8)	1 Mbyte	42.57

Fig. 2. Timing Results

These can be compared to the `SHA-1` based implementation, which took 1.22 seconds to produce a similar 863 Kbit block of pseudo-random data. This implementation is written in highly optimised `C` code; in fact this is the code that the CP/Q operating system itself uses to generate pseudo-random data. However we do note that we ran the code as a standard "loaded-in" application, just as the number theoretic generator was, to enable a fair comparison.

Another useful comparison is to the BBS generator (see [2]), where one obtains at least 1 bit (and at most[2]) about 4 bits) of pesudorandom data from

[2] This has to do with assumptions on the hardness of factoring; see [5] for more details.

each modular squaring. Theoretically this should be similar to our exponentiation method (see [5] for a more rigorous comparison), however the overhead of calling the modular math hardware adversely affects this generator. In fact it takes 2.3 seconds to generate just a 1Kbit block of data using this approach (assuming one bit per exponentiation).

5 Conclusions

The results show that the `SHA-1` based pseudorandom number generation is still considerably faster than the one based on discrete logarithms. However the difference, a factor of less than 4 on this hardware, may be considered not too high a price to pay by some who wish to have a "provably secure", rather than a "seemingly secure" (i.e. one that has withstood cryptographic attack thus far) system for pseudorandom number generation.

It should be stressed however that this result is strongly reliant on the fact that the algorithms were tested on the IBM 4758 secure coprocessor, which has support for hardware modular exponentiation. All of the software-based exponentiation variants of [5] that we tried were considerably slower (another factor of 10 to 20), even though they made use of hardware support for modular multiplication, and used precomputed tables.

The discrepancy was even more significant with the BBS generator due to the low output rate of the generator for each call to the modular math hardware; it turned out to be between 100 and 400 times slower than the "pure exponentiation" generator on this hardware.

References

1. L. Adleman. *A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography.* IEEE FOCS, pp.55-60, 1979.
2. L. Blum and M. Blum and M. Shub *A Simple Unpredictable Pseudo-Random Number Generator.* SIAM J.Computing, 15(2):364–383, May 1986.
3. M. Blum and S. Micali. *How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits.* SIAM J.Computing, 13(4):850–864, November 1984.
4. W. Diffie and M. Hellman. *New Directions in Cryptography.* IEEE Trans. Inf. Theory, IT-22:644–654, November 1976.
5. R. Gennaro. *An Improved Pseudo-random Generator Based on Discrete Log.* CRYPTO'2000, LNCS 1880, pp.469–481, 2000. Updated version available at http://www.research.ibm.com/people/r/rosario/prng.ps
6. D. Knuth. *The Art of Computer Programming (vol.3): Sorting and Searching.* Addison-Wesley, 1973.
7. C.H. Lim and P.J. Lee. *More Flexible Exponentiation with Precomputation.* CRYPTO'94, LNCS 839, pp.95–107.
8. National Institute of Standards and Technology. FIPS 140-1, *Security Requirements for Cryptographic Modules.* Available at http://csrc.nist.gov/cryptval/140-1.htm
9. S.C. Pohlig and M.E. Hellman. *An Improved Algorithm for Computing Logarithms over $GF(p)$ and its Cryptographic Significance.* IEEE Trans. Inf. Theory, vol.IT-24, no.1, p..106–110, January 1978

10. J. Pollard. *Monte-Carlo Methods for Index Computation (mod p).* Mathematics of Computation, 32(143):918–924, 1978.
11. C. Schnorr *Security of Allmost ALL Discrete Log Bits.* Electronic Colloquium on Computational Complexity. Report TR98-033. Available at `http://www.eccc.uni-trier.de/eccc/`.
12. S. Smith and S. Weingart. *Building a High-Performance, Programmable Secure Coprocessor.* Special Issue on Computer Network Security, Elsevier, 1990, v.31, pp 831-860. Also, IBM Research Report RC21102, February 1998.
13. P.C. van Oorschot and M. Wiener. *On Diffie-Hellman Key Agreement with Short Exponents.* EUROCRYPT'96, LNCS 1070, pp.332–343, 1996.
14. A. Yao. *Theory and Applications of Trapdoor Functions.* IEEE FOCS, 1982.