Pseudorandom Number Generation in Smart Cards: An Implementation, Performance and Randomness Analysis

Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes
Information Security Group, Smart Card Centre, Royal Holloway, University of London.
Egham, United Kingdom. Email: {R.N.Akram, K.Markantonakis, Keith.Mayes}@rhul.ac.uk

***Abstract*—Smart cards rely on pseudorandom number generators to provide uniqueness and freshness in their cryptographic services i.e. encryption and digital signatures. Their implementations are kept proprietary by smart card manufacturers in order to remain competitive. In this paper we look at how these generators are implemented in general purpose computers. How architecture of such generators can be modified to suit the smart card environment. Six variations of this modified model were implemented in Java Card along with the analysis of their performance and randomness. To analyse the randomness of the implemented algorithms, the NIST statistical test suite is used. Finally, an overall analysis is provided, that is useful for smart card designers to make informed decisions when implementing pseudorandom number generators.**

***Keywords*-Smart Cards; Pseudorandom Number Generators; NIST Statistical Test 800-22; Performance Measurements;**

I. Introduction

In the last decade, besides the use of the smart card technology in the banking and the telecommunication industries, such technology gained acceptance in other fields such as security tokens (keys), driving licenses, national ID cards, student ID cards, social security, health, travel documents, and passports. In most of these fields, smart cards use applications that require generation of session keys, DSS signatures, and pseudorandom padding bits, etc. These functionalities require a secure and unpredictable pseudorandom number generator for their security [1].

This paper examines what is randomness and how the general-purpose computers generate it. It also looks at security requirements that a Pseudorandom Number Generator (PRNG) has to satisfy for use in cryptographic applications. A generic model for implementing a PRNG is analysed along with six variations of the model is implemented on smart cards. These implementations are evaluated on their performance and subjected to randomness tests.

The structure of the paper is as follows: section two introduces pseudorandom numbers, their implementation methodology in terms of general-purpose computers and their desirable properties. Furthermore, it describes the evaluation criteria for randomness of the generator with the NIST SP 800-22 statistical test suite [2]. Section three explains the general architecture of smart cards and the constraints on the smart card based PRNG. The generic model is described in section four, along with the implementation details for each of the six algorithms. Theoretical and experimental proof of their security and randomness is illustrated in section five. Finally, section six offers an overall analysis of the research and of its outcomes.

A. Motivation

As most cryptographic mechanisms requires some sort of input from a random number generator. A weakness in the generator will eventually weaken the cryptographic mechanism. If the generator is weak, it can cause the entire mechanism to be insecure. This is evident from the results shown in the reverse engineering of the Mifare Classic [3], [4], [5], [6].

For a generator to be cryptographically secure the generated sequences should be random that an adversary is unable to predict future or determine past sequences. A preferable choice is to have hardware based True Random Bit Generator (TRNG) [7], [8], [9]. However, on a resource restrict device like smart cards, having TRNG may be vulnerable to the differential fault analysis [10], [11]. Furthermore, a number of standards have stringent requirements for a TRNG. For example, in the Common Criteria [12] (German scheme) the requirements for the TRNG are defined in the AIS31 [13]. It states that on each activation of the generator, tests specified in the FIPS 140-2 [14] should be performed. This impose a huge computational cost on the TRNG mechanism and it might also adversely affect the security of the generator against attacks like SPA/DPA [15] and template attacks [16]. As discussed by the [17] that these stringent requirements make it difficult to have an efficient and usable TRNG. Therefore, the alternative would be to have a Pseudorandom Number Generator (PRNG). A PRNG implementation in computers are usually tested using number of statistical tests (like in the NIST SP 800-22 [2] test suite there are 15 tests). However, the Common Criteria (German Scheme) details the list of test in AIS20 [18] which consist of only five tests. The reason behind this may be smart card's restricted resources. However, to have an assurance that a PRNG in smart cards is equally produce random sequences as a computer based PRNG. Smart card based PRNG should also be rigorously tested using the same statistical test batteries as a computer based PRNG. In this paper, we provide six PRNGs based on different cryptographic constructs along with memory usage and performance measurements. In addition, all of these PRNGs are tested using the NIST SP 800-22 test suite.

II. Pseudorandom Numbers

In this section, we first explain pseudorandom numbers, followed by how they are generated, and finally the description of the NIST statistical test suite.

A. Pseudorandom Number Generator

A PRNG defined by [1] is an "algorithm which given a truly random binary sequence of length k, output a binary sequence of length l >>k which appears to be random". According to this definition, on the elementary level, a generator consists of a function called a generation algorithm that takes a small random string and converts it into a longer string that is statistically independent of the input string. Input to a generation algorithm is known as the seed and it is critical to the randomness of the generator's output. The seed values are taken from an entropy source where entropy is a measure of disorder, randomness, or variability in a closed system. The randomness or disorder is relative to the observer, if the observer is unable to predict the scale of disorder, then the entropy source provides high entropy [2]. Figure 1 shows an elementary level model of a PRNG.

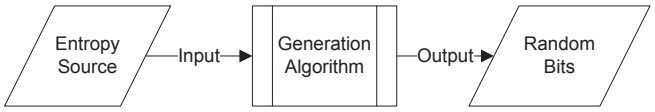

Fig. 1. Generic Model for PRNG

Additionally, PRNG should also have a one-way property, making it computationally difficult to find the input string from the generated output sequence. Finally, the overall model of a PRNG should have the desirable properties that are explained in next section. There are well defined standards such as NIST SP 800-90 [19] and ISO/IEC 18031 [20] that provides the guideline on how a PRNG can be implemented in the computer environment. In this paper, we follow these guidelines and only changing/modifying when required or compelled by smart card environment.

B. Desirable Properties of Pseudorandom Number Generator

There are five properties that are desirable in any implementation of PRNG [5]. The first property is a functional one and other four are security properties.

1) PRNG outputs should be computationally indistinguishable from TRNG.
2) Backtracking Resistance: Even after the compromise of the internal state, an adversary will not be able to generate past output. The PRNG's internal state is actually its memory. The values stored in it may be counter, user input, predefined system values and cryptographic keys, etc. All these values are used during the generation process, and they serve as input to the generation algorithm along with the seed.
3) Prediction Resistance: An algorithm provides prediction resistance if an adversary is unable to predict future outputs, even after the compromise of the internal state of the generator. For this, a generator should regularly be reseeded. Reseeding mean changing the entropy sources on detection of a compromise or at the end of its lifetime.
4) Backward Secrecy: An adversary is not able to predict past outputs, even if s/he observes all future outputs from a given point of time. His/her knowledge about the generated sequences is restricted to the first observation and all previous outputs will always remain hidden.
5) Forward Secrecy: An adversary will still not be able to predict future outputs, even after observing past outputs. The probability of predicting future values should not be increased by observing the outputs over time.

Statistical tests verify the first desirable property by measuring the level of randomness in the generated output. Several statistical test suites are available such as Diehard [6], TestU01 [3] and NIST SP 800-22 [2] that only give an assurance that there are no repeated patterns in the bit sequence generated. However, no set of statistical tests can absolutely certify a generator as being appropriate for usage in a particular application, i.e. statistical testing cannot serve as a substitute for cryptanalysis [2].

C. NIST SP 800 - 22

For evaluation of randomness in a generated sequence, there is no uniform methodology. However, several statistical test suites are available to perform this task. Among them, we have chosen the NIST statistical test battery for the following reasons. Firstly, it contains many famous tests from the Diehard battery with some extra tests. Secondly, It was used in the AES evaluation process to check the randomness of the output sequences of each candidate algorithm.

The NIST SP 800-22 consists of the tests mentioned in Appendix B, along with their input parameters used to evaluate sequences generated by the implemented algorithms. After pseudorandom sequences generated by the PRNG were subjected to NIST statistical tests, the results were interpreted in two ways, as recommended by [2]. The proportion of sequences passing test that analyses how many output sequences pass a particular test out of total sequences generated by the algorithm. The second test is uniform distribution of P-value that analyses the distribution of passing sequences in the output to observe any irregularity. Both methods analysed the test results on the generated output to see the ratio and distribution of passing sequences.

III. Smart Card Technology

In this section, we review the smart card technology and its implications on the PRNG design.

A. Smart Card Hardware

Smart card is a resource-scarce platform that typically consists of a central processing unit (CPU), read only memory (ROM), random access memory (RAM), electrical erasable programmable read only memory (EEPROM), and a Crypto-Coprocessor [21]. In the smart card, data can be stored in ROM

and EEPROM. ROM houses the Card Operating System and EEPROM stores the data that needs to be updated over the operational lifetime of the card. The EEPROM memory has restricted write/erase cycles, and it ranges between 100,000 to 1,000,000 cycles over the entire range of operating temperatures and voltages [1], after that it becomes inoperable.

For its entire lifetime, the smart card remains in the possession of the cardholder [22]. This allows an adversary to perform attacks on the card hardware or software security without any physical access restriction on it. As essential data is stored in EEPROM, if there is no protection mechanism available, an adversary can effectively change the values stored in EEPROM to his/her choice. However, cards provided by a respectable manufacturer will offer adequate security measures to prevent such attacks.

B. Smart Card Platform Constraints on PRNGs

For any PRNG, a random source of entropy is required, from which the generation algorithm takes a seed value and generates a pseudorandom number. Such an entropy source in PCs can be typing speed, noise recorded by a microphone, processes in queues and their time/status.

However, in the smart card, entropy sources are limited and the card cannot rely either on an internal mechanism to generate or on external source to provide the entropy required by the PRNG. Besides the problem of a good entropy source, smart cards are also constrained by the amount of memory and by processing capabilities. Therefore, an ideal solution would be a hardware-based PRNG. Current random number generators in the smart card micro-controllers are generally based on Linear Feedback Shift Registers (LFSR) driven by voltage-controlled oscillators [1].

However, sequences produced purely by LFSRs have a high level of linearity, which makes it easy for an adversary to guess internal values and to predict future outputs. Therefore, for cryptographic use they do not provide the required level of security [23]. Besides the hardware-based models, a PRNG can also base purely upon software implementations. However, the choices of these are restricted in smart cards due to: 1) Limited memory capacity and processing speed of the smart card. 2) Non-availability of good entropy sources and 3) Reseeding is not implemented for security reasons. Finally, 4) FIPS 140-2 [14] restrictions.

Technically, reseeding the smart card is possible and the motivation to do so is to provide prediction resistance to the PRNG. This can be implemented by requesting reseeding when smart card is inserted into the card reader. However, if the provision for changing the data in the generator is provided after the card manufacturing, then an adversary may attack this mechanism to his/her advantage, enabling him/her to predict future values. As a result, not enabling an adversary to provide or manipulate the input to the PRNG prevents the retrieval of the internal state of a generator.

IV. Pseudorandom Number Generators in Smart Cards

In this section, we present a generic model for PRNG and details of implemented algorithms based on the generic model along with the performance measurement.

A. Generic Model of Pseudorandom Number Generator

Keeping under consideration the desirable properties and model guidelines mentioned in Section 2.2. Adopting and in some cases modifying the design recommendations of [1, 11, 17, 16, 19, 20] to suit the smart card environment, we have put forward a generic model as illustrated in figure 2. The chosen architecture of the smart card PRNG is a slight modification of the [14, 21, 22].This, with minimal modification, can be implemented in most commercially available smart cards.

Fig. 2. Generic Model for Pseudorandom Number Generator in Smart Cards

1) The Input Formatting Function (IFF) retrieves the seed value from the Cyclic Buffer (CB). The CB provides the entropy source to the PRNG in smart cards. Typically a CB (also can be referred as seed file) stores ten random seed values and each value is used/updated sequentially. The IFF processes the data to meet the input requirements of the Generator Algorithm (GA). The IFF keeps the record of which value to fetch on each execution and this value is referred as the seed index.
2) The output generated by the IFF is processed by the Generator Algorithm (GA). A GA is actually a cryptographic function (i.e. Hash [23], MAC [23], DES [24], and AES [25] etc). The choice of the GA is with the smart card developers (or implementers).
3) The output of the GA is fed to Seed Update Generator (SUG). The SUG will generate the value that is used in the seed update process and also back to IFF. The basic architecture of SUG is that is Xor the input to GA with its output. If the input/output mismatch in length, then it concatenate the shorter length with itself until it is equal to the other one.
4) The output of the SUG is processed by the Seed File Update Function (SFUF). The SFUF fetches the update value (to input seed) and Xor with the SUGs output. The result is written back to the same location from where the SFUF fetches the value. The update value can be seed index 1 and this method is used in all of the implementations in this paper.
   The output of the SUG is fed back to the IFF for the second round. In the second round the step 1 and 2 are performed again. However, the output of the GA is not sent to SUG, instead it goes to Output Formatting Function (OFF). This additional iteration of step 1 and 2 is performed mainly to conceal the value that is used to update the seed file (Cyclic Buffer).
5) The Output Formatting Function receives an input from the GA and it formats this output to the requirements stipulated by the requesting user or application. Once it formats the input to desired output, it sends the pseudorandom number to requesting entity.

B. Implemented Algorithms

Based on the generic model for Pseudorandom Number Generator (PRNG) described in the previous section, six implementations with varying Generator Algorithms (GAs) were done. These implementations were not just the change in the GA but also the related functions shown in the figure. The implemented algorithms are illustrated in Appendix A. Table I is the list of cryptographic algorithms used as the GAs, input seed requirements, generated pseudorandom sequence length [19].

TABLE I
ALGORITHMS WITH INPUT AND OUTPUT LENGTH

Generator Algorithm	Input Length (bits)	Output Length (bits)
SHA - 1	440	128
SHA - 256	440	128
HMAC - DES	128	128
HMAC - AES	128	128
DES	128	128
AES	128	128

For the experiments, all implementations were in Java Card. For initial testing, we implemented them on a simulator (Java Card simulator JCWDE [26]) and later on live cards. The JCWDE provide support for all of the algorithms in the above-mentioned table. The test environment was set on a 1.8 GHz PC with 512 RAM. An application that requested pseudorandom sequences from PRNG installed in JCWDE and also from the live cards was developed in Java 1.5 and it stored the generated results to a binary file. To verify the accuracy of the sample taken from the Java Card simulator, we also conducted tests on the live cards and found out that the results were the same. For this comparison, we used the same seed file for both the simulator environment and live card PRNG.

The following table shows the size of each implementation in EEPROM. In commercial implementation, only the seed file is in EEPROM and not the code of the PRNG which would be part of the Smart Card Operating System and resides in ROM. The values shown in table II are code and seed file sizes. The proposed model works with any size of seed file as long as it provides high entropy. However, for our experiments we used the seed file of the size mentioned in the following table.

TABLE II
ALGORITHMS WITH INPUT REQUIREMENTS AND PRODUCED OUTPUT LENGTH

Generator	Program Size	Seed File Size
SHA - 1	2086 bytes	550 bytes
SHA - 256	2123 bytes	550 bytes
HMAC - DES	2283 bytes	160 bytes
HMAC - AES	2299 bytes	160 bytes
DES	2245 bytes	160 bytes
AES	2189 bytes	160 bytes

C. Pseudorandom Number Generator's Performance Analysis

Once the algorithms were implemented on the live cards, we conducted the performance tests. Before going into the details of these tests, one thing to note is that these tests were conducted on the PRNG that were implemented in the Java Card language [26] running over the Java Card Virtual Machine [27]. This architecture cannot be considered the most efficient way of implementing the PRNG. The PRNGs are normally implemented in the native code by the smart card manufacturer, so their performance will always be better then our implementations. The reason for listing the performance matrix in the paper is to present that a PRNG implemented above the Smart Card Operating System (SCOS) [1] layer can still be viable and perform efficiently. The performance matrix is the table that is used to record the measurements of the algorithm execution.

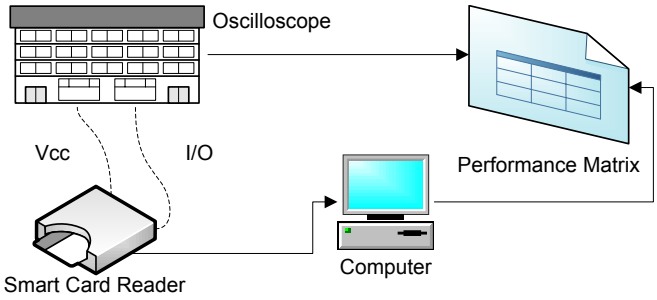

Fig. 3. Performance Measurement Framework

The basic framework to measure the performance of a PRNG is illustrated in the figure 3. The smart card is inserted in to a smart card reader that is connected to an oscilloscope. A test application executes a script that selects a PRNG and then request 128 bits of random number. The time a smart card takes from receiving the request till producing an output is calculated from the oscilloscope and the means value is taken for the performance measurement.

For performance analysis we selected two 16-bit Java Cards (from different manufacturers) and implemented the PRNG algorithms (at application layer) based on the design illustrated in figure 2 with adequate modifications (see Appendix A). Each algorithm has a performance matrix that is generated by 1000 measurements from each card. Table III illustrates that the fastest PRNG implementation on a smart card are Hash based generators. This table does not depicts the best PRNG as in addition to the performance each PRNG should also be evaluated for their randomness and security properties before making the decision to use it in any application. The NIST Statistical Test Suit SP 800-22 is used for the evaluation of randomness in the generated output. The details of the statistical tests for each of the algorithms are listed in the table IV.

D. Theoretical Proof

Proof that the algorithms based on the proposed model satisfy the desirable security properties of PRNG mentioned in 2.2 is as follow:

1) Backtracking Resistance: Basing the model on a well-established cryptographic principle ensures that even after gaining knowledge of the inner state, an adversary still cannot predict previous outputs. As Hash functions are inherently irreversible, where in MAC and symmetric functions, knowledge of the keys used is essential

TABLE III
PERFORMANCE MEASURE (MILISECONDS)

Performance Measure		SHA-1	SHA-256	HMAC-DES	HMAC-AES	DES	AES
Card One	*Average Time*	43.85	48.39	272.29	273.29	156.56	154.98
	Fastest Time	40.27	44.83	267.80	269.17	151.76	150.00
	Slowest Time	55.57	53.39	287.89	282.80	168.19	165.26
	Standard Deviation	4.48	3.13	5.12	4.22	4.21	4.50
Card Two	*Average Time*	42.84	47.64	263.87	263.02	156.56	148.02
	Fastest Time	39.45	44.08	257.32	257.32	151.76	144.54
	Slowest Time	54.01	58.86	272.17	272.11	168.19	152.01
	Standard Deviation	4.40	4.23	3.57	3.70	4.21	2.37

in backtracking. Which is computationally equivalent to finding pre-image or exhaustive key search respectively.

2) Prediction Resistance: To ensure that an algorithm is prediction resistant, regular reseeding is required, which is a non-preferable operation in smart cards. For this reason, our implementations provided limited or no prediction resistance. This is not due to the model of the algorithms but due to the general security principle adopted in smart cards.
3) Backward Secrecy: All of our algorithms are based around strong cryptographic principles that inherently provide backward secrecy. To compromise our proposed model, an adversary first has to break them e.g. SHA-256, AES, which is computationally equivalent to finding a pre-image in the case of hash functions or an exhaustive key search.
4) Forward Secrecy: To provide forward secrecy, the number of times a PRNG can execute should have a limit. This limit is left to the smart card developer and the intended application use.

There are special attacks on the smart card-based cryptographic algorithms known as side channel attacks [15]. They make even the best algorithms (AES) weak by reducing the complexity of a key search drastically. These attacks gather a large number of observations and analyse them statistically. If all of these observations are generated by a single key, then there is a high probability that side channel attacks can recover it. However, as our implementations use a new key per execution without affecting the overall randomness of the generated output. The side channel attacks may not be as successful as in previous case. Any algorithms presented cannot remain secure because they are based on cryptographic principles. Preventive measures with regard to these attacks also have to be implemented at the hardware and software code levels. Therefore, at model level there is not much we can do to prevent such attacks. Thus, the protection of the proposed algorithms from these attacks is based on the overall security of the smart card hardware protection mechanism and on secure coding practice.

E. Experimental Proof

To provide experimental proof, the NIST statistical test suite was applied. Each algorithm was provided with a common seed file, and generated sequences from it were saved in a binary file. This binary file was used as input to the statistical test. Point to note here is that seed files given to all algorithms were the same. The reason for doing so was to analyse differences in the quality of output while using the same entropy source.

For statistical analysis, each algorithm was executed to generate 1,048,578 pseudorandom sequences of 128 bits. Concatenating the outputs into a binary file that was then used for NIST SP 800-22 statistical analysis. The results of each algorithm are listed in Appendix A. Taking into account the Common Criteria AIS 20 [18], our implementation fulfils the requirements for the K4 DRNG. Below is the discussion on how our implementation satisfies these requirements.

1) K1 DRNG: Its a simple requirement that states that if the generated values is of set C =$\{$ c_1, c_2, c_3, .., c_m $\}$ then all members of the set should be distinct regardless of the statistical properties.
2) K2 DRNG: Requires that the implementation should satisfy the statistical properties such as monobit test, poker test and tests on runs. Our implementations were subjected to the NIST SP 800-22 test suite.
3) K3 DRNG: This requires that the entropy of the PRNG is at least 80. All SHA based algorithms has 440 bits seed and block cipher based algorithms has 128 bits seed. All of the seed values were chosen from an external high entropy source that is carefully tested.
4) K4 DRNG: This level requires that the PRNG should be forward-secure. A PRNG is forward-secure if after n iteration of the PRNG, a malicious user is unable to guess the internal state of the generator. The implemented PRNG feed back to the internal seed that is changing in each of the iterations. Furthermore, block cipher based implementation of the PRNG use different key in each of the iterations. Even retrieving a cryptographic key would not help a malicious user to successfully know the entire state of the seed file.

In our implementations we tested SHA and block cipher based algorithms for the PRNGs. In general block cipher based algorithms, only a single key is used for the entire lifetime of the generator. However, we have tested that a PRNG could be modified to use a new key on each execution of the generator. The internal key generation mechanism of the block cipher based PRNG implementations are light weight and they do not hamper the overall performance of the generator. Therefore, it could be argued that it provides a more secure block cipher based PRNG then an PRNG that only uses a single key for entire lifetime.

Table IV details the percentage of passing sequences pro-

TABLE IV
NIST STATISTICAL TEST RESULTS (PERCENTAGE OF PASSING SEQUENCES)

Algo / NIST Tests	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SHA-1	98.60	98.99	99.00	98.00	100	99.04	100	100	100	98.00	100	99.00	98.81	99.47
SHA-256	99.30	100	97.00	97.00	98.97	98.78	100	98.00	99.00	99.50	99.00	98.00	98.86	100
HMAC-DES	99.20	100	97.00	99.00	98.97	99.04	99.00	100	98.00	100	100	98.00	100	97.62
HMAC-AES	99.30	100	99.00	100	100	99.01	98.00	97.00	97.00	99.50	94.00	98.50	99.04	100
DES	99.20	98.99	100	100	100	97.98	100	99.00	100	99.00	97.00	99.00	96.67	96.67
AES	99.20	97.98	96.00	97.00	98.97	97.98	98.00	99.00	99.00	100	96.00	98.50	100	100

duced by individual algorithms. As it is evident from table III and IV, there is not a big difference between the implemented algorithms both in terms of performance and percentage of sequence passing the NIST statistical tests. Of particular note, if we take the accumulated average of the passing sequences percentage in the table IV an interesting result emerges. The SHA-1 performs comparative better than other algorithms and AES based PRNG has the least accumulated average of passing sequence as illustrated in figure 4. This measure represents the randomness of the generated sequences - not the security of the algorithm. If we also account for the performance, SHA based algorithms perform better than the encryption based algorithms (e.g. DES and AES).

V. CONCLUSION AND FUTURE RESEARCH DIRECTIONS

All of the algorithms proposed in this paper used hardware-based cryptographic primitives as GAs. While PRNGs can be entirely software-based, the resource-restricted environment and FIPS 140-2 requirements [14] compelled the model to be partially based on hardware implementations.

Another deviation was from the standard method of implementing PRNG from hash and block cipher-based algorithms given in NIST SP 800-90 [19] and ISO/IEC 18031[20]. We only consider one proposal to state our reason for not following it faithfully. The hash-based PRNG presented by NIST [19] contains three iterations of hash function. This would increase the time of execution without giving any substantial increase in randomness. The inclusion of additional input to the process is not viable in the smart card environment, because there is not enough random or diverse information available for this purpose. As reseeding in the smart card is not preferred, there was no need to include a reseeding counter, while a simple counter was not implemented due to the limited write cycles of the EEPROM memory.

Our research into the possibility of using a test suite like NIST SP 800-SP to check pseudorandom number generators in smart cards has showed that it is a workable concept. The tests listed in the NIST SP 800-22 are substantially more then the one recommended for the smart card pseudorandom number generators in AIS20 [18] and AIS31 [13]. This research has demonstrated that even with limited resources and an entropy constrained environment like a smart card, good quality pseudorandom sequences can be generated that can satisfy all the requirements for a PRNG even the ones that are used for general purpose computers.

We would like to extend the statistical test suite to include other tests that are not included in the NIST statistical suite and analyse whether the implemented algorithms satisfies them or not. Due to unavailability of native implementation of different algorithms (e.g. MD5 and T-DES) in the test cards, we could not test them for their randomness by generating sequences on Java Card simulator; however, lack of performance measure on an actual smart card prohibited us from including them to the analysis. Therefore, we would like to incorporate these algorithms along with testing the algorithms from public key cryptography (e.g. ECC) to illustrate their performance in the smart card environment.

VI. ACKNOWLEDGEMENTS

We would like to extend our appreciation to the anonymous reviewers for their valuable time and feedback. Additionally, thanks to Min Chen, Babar Mahmood, and Hisham Abbasi for patience while proof reading drafts.

REFERENCES

[1] W. Rankl and W. Effing, *Smart Card Handbook*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[2] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "NIST SP 800-22: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Application." National Institute of Standards and Technology, U.S. Department of Commerce, NIST SP 800-22, May 2001.

[3] G. Koning Gans, J.-H. Hoepman, and F. D. Garcia, "A Practical Attack on the MIFARE Classic," in *CARDIS '08: Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*. Springer, 2008, pp. 267–282.

[4] F. D. Garcia, G. de Koning Gans, R. Muijrers, P. van Rossum, R. Verdult, R. W. Schreur, and B. Jacobs, "Dismantling MIFARE Classic," in *ESORICS*, 2008, pp. 97–114.

[5] K. Nohl, D. Evans, S. Starbug, and H. Plötz, "Reverse-Engineering a Cryptographic RFID Tag," in *SS'08: Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 185–193.

[6] F. D. Garcia, P. v. Rossum, R. Verdult, and R. W. Schreur, "Wirelessly Pickpocketing a Mifare Classic Card," in *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 3–15.

[7] E. Trichina, M. Bucci, D. De Seta, and R. Luzzi, "Supplemental Cryptographic Hardware for Smart Cards," *IEEE Micro*, vol. 21, no. 6, pp. 26–35, 2001.

[8] M. Bucci, L. Germani, R. Luzzi, A. Trifiletti, and M. Varanonuovo, "A High-Speed Oscillator-Based Truly Random Number Source for Cryptographic Applications on a Smart Card IC," *IEEE Trans. Comput.*, vol. 52, no. 4, pp. 403–409, 2003.

[9] I. Vasyltsov, E. Hambardzumyan, Y.-S. Kim, and B. Karpinskyy, "Fast Digital TRNG Based on Metastable Ring Oscillator," in *Cryptographic Hardware and Embedded Systems – CHES 2008*, ser. LNCS, E. Oswald and P. Rohatgi, Eds. Springer, August 2008, vol. 5154, pp. 164–180.

[10] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *CRYPTO '97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*. London, UK: Springer, 1997, pp. 513–525.

[11] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," in *EUROCRYPT'97: Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*. Springer, 1997, pp. 37–51.

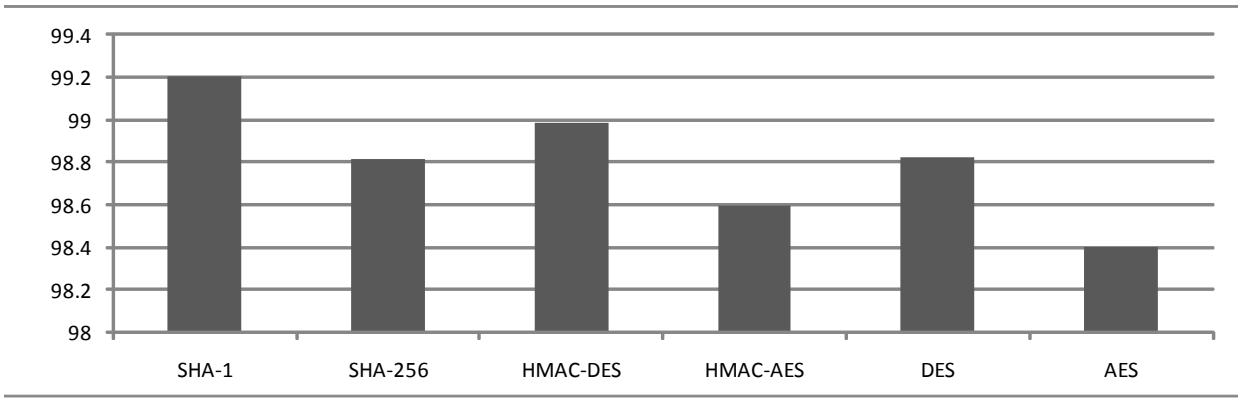

Fig. 4. Graph Depicting Average Percentage of Passing Sequence Across NIST Statistical Test Results (Table IV)

[12] *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model, Part 2: Security functional requirements, Part 3: Security assurance requirements,*, Common Criteria Std. Version 3.1, August 2006. [Online]. Available: http://www.commoncriteriaportal.org/thecc.html

[13] "BSI AIS 31: Functionality classes and evaluation methodology for deterministic random number generators," Certification body of the BSI as part of the certification scheme version 2, September 2001. [Online]. Available: https://www.bsi.bund.de/cae/servlet/contentblob/478130/publicationFile/30547/ais31e_pdf.pdf

[14] (2001, May) FIPS 140-2: Security Requirements for Cryptographic Modules. National Institute of Standards and Technology. Washington, DC.

[15] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," *Lecture Notes in Computer Science*, vol. 1666, pp. 388–397, 1999.

[16] S. Chari, J. R. Rao, and P. Rohatgi, "Template Attacks," in *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*. London, UK: Springer-Verlag, 2003, pp. 13–28.

[17] S. Chari, V. V. Diluoffo, P. A. Karger, E. R. Palmer, T. Rabin, J. R. Rao, P. Rohatgi, H. Scherzer, M. Steiner, and D. C. Toll, "Designing a Side Channel Resistant Random Number Generator," in *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010*, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds. Springer, April 2010, pp. 49–64.

[18] "BSI AIS 20: Functionality classes and evaluation methodology for deterministic random number generators," Tech. Rep. version 2, December, 1999. [Online]. Available: https://www.bsi.bund.de/cae/servlet/contentblob/478152/publicationFile/30552/ais20e_pdf.pdf

[19] E. Barker and J. Kelsey, "NIST SP 800-90: Recommendation for Random Number Generation Using Deterministic Random Bit Generators," National Institute of Standards and Technology, U.S. Department of Commerce, Tech. Rep., March 2007.

[20] "ISO/IEC 18031: Information Technology-Security Techniques-Random bit generation," *International Organization for Standardization and International Electrotechnical Commission*, vol. iso 18031, 2005.

[21] K. Mayes and K. Markantonakis, Eds., *Smart Cards, Tokens, Security and Applications*. Springer, 2008.

[22] R. N. Akram, K. Markantonakis, and K. Mayes, "A Paradigm Shift in Smart Card Ownership Model," in *Proceedings of the 2010 International Conference on Computational Science and Its Applications (ICCSA 2010)*, B. O. Apduhan, O. Gervasi, A. Iglesias, D. Taniar, and M. Gavrilova, Eds. Fukuoka, Japan: IEEE Computer Society, March 2010, pp. 191–200.

[23] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[24] *DES: Data Encryption Standard*, ser. Federal Information Processing Standard Publication. Washington : The Bureau ; Springfield: U.S Department of Commerce, National Bureau of Standards, January 1977.

[25] J. Daemen and V. Rijmen, *The Design of Rijndael*. Springer, 2002.

[26] Z. Chen, *Java Card Technology for Smart Cards*. Addison-Wesley, 2000.

[27] *Java Card Platform Specification; Application Programming Interface, Runtime Environment Specification, Virtual Machine Specification*, Sun Microsystem Inc Std. Version 2.2.2, March 2006. [Online]. Available: http://java.sun.com/javacard/specs.html

APPENDIX

Following table shows the list of tests from NIST statistical test suit and the input parameters mentioned calculated according to [2] guideline.

TABLE V
NIST SP 800 - 22 STATISTICAL TESTS INPUT PARAMETERS

Statistical Test	**Outputs**	**Output Size**
1. Frequency Test (Monobit)	1000	2000
2. Frequency Test (Block)	99	12672
3. Runs Test	100	16384
4. Longest Run of Ones	100	750000
5. Binary Matrix Rank Test	97	38912
6. Non-Overlapping Template	99	131072
7. Maurers Universal Statistical	100	1048576
8. Overlapping Template	100	1000000
9. Linear Complexity Test	100	1048576
10. Serial Test	100	1048576
11. Approximate Entropy Test	100	1048576
12. Cumulative Sums (Cusum) Test	100	128
13. Random Excursions Test	100	1048576
14. Random Excursion Variant Test	100	1048576