Subverting the NIST Hash PRNG

Mihael Rajh

Faculty of Computer and Information Science
Večna pot 113
1000 Ljubljana

Abstract. The study of secure information theft known as kleptography has, since its introduction in the 90s, been applied to a large variety of cryptographic protocols. In this project report we focus on pseudo-random number generators (PRNGs), which are a common requirement in cryptography. We implement the NIST Hash DRBG in Python and attempt to subvert it using a substitution attack based on rejection sampling. While the attack is weak, it is largely undetectable due to the use of legitimate operations. This was confirmed through testing with the NIST Statistical Test Suite. On the other hand, an attempt to improve the attack by increasing its throughput was largely ineffective. In order to protect against substitution attacks, NIST mandates the use of internal testing, which was found to be easily subverted. However, the reseeds mandated by prediction resistance were not, and effectively guard against our attack. We hope that the study of kleptography and defences against it continues to evolve.

Key words: kleptography, algorithm substitution attacks, pseudo-random number generators, rejection sampling

1 INTRODUCTION

As the amount of information transferred and stored by technology has increased, so has the need to protect it through suitable cryptographic systems. Data breaches are among those attacks that are only becoming more frequent with time [15]. Furthermore, it is sometimes possible to steal information not only subliminally, but also securely, meaning that it can be recovered exclusively by the attacker. This is the setting that is studied by the field of kleptography.

Introduced in 1996 [18] and 1997 [19] by Young and Yung, this idea has since proliferated and been applied to a large variety of cryptographic primitives. The core of the approach is to use existing or artificially created subliminal channels of a cryptographic process in order to leak information. This is done by substituting the legitimate implementation in a way that is not detectable by the user.

The authors who coined the term have demonstrated efficient attacks on the Diffie-Hellman key exchange [19], with focus on ElGamal and RSA key generation [18] [20], and application in SSL [22] [21]. Other authors have studied similar approaches under the name of algorithm substitution attacks (ASAs) for symmetric encryption [5] [4]. The idea became more than theoretical with the standardization of a backdoored cryptographic random number generator, the Dual EC Deterministic Random Bit Generator (DRBG) [8], which has since been removed from valid recommendations.

This paper focuses on the exploitation of pseudo-random number generators (PRNGs). While the output of such constructs is usually fed into consuming applications, a poor source of randomness can contribute to a wide range of vulnerabilities [9]. Additionally, the output of a PRNG does not have to be decryptable, only difficult to predict. We implement the Hash DRBG, standardized by NIST [2], in order to investigate kleptographic concepts and defences against them.

In the second chapter, an overview of kleptography and PRNGs is given, together with observations on some related key concepts. Next, we describe the structure of the NIST Hash DRBG, the implemented kleptographic attack, and the NIST Statistical Test Suite. In the fourth chapter we provide the details of our PRNG implementation and some statistical test results. We follow this with an evaluation of detectability and practicality of the implemented attack and defences against it. Finally, we give a summary and our conclusions.

2 THEORETICAL BACKGROUND

Adversarial attacks can be said to form the very core of cryptographic mechanisms, as the latter are designed to counter threats to information privacy, integrity, and authenticity among others. However, kleptographic tactics typically fall outside of such threat models [8], since they do not rely on breaking cryptographic mechanisms, but rather sabotaging them. In order for such an attack to be successful, the perpetrator must be much more powerful than what is typically considered.

Most cryptographic threat models consider adversaries that can listen to, modify, or replay messages, while the kleptographic threat model considers adversaries that can modify the mechanisms in use. Substantially more effort is required for this, and the process would likely give direct access to whatever information is desired, unless it has not yet been processed. Because of this, the central concern in kleptography is typically not an invasion of privacy by malicious individuals, but rather large-scale surveillance by governments or large corporations, sometimes termed ”big brother” [5].

In this section, we first describe the basic principles on which kleptographic attacks operate, and provide an overview of some successful approaches to algorithm subversion. We also observe why such approaches are effective. We then move on to PRNGs, giving examples of how faulty PRNGs have be exploited, what it means for a PRNG to be cryptographically secure, and their relation to encryption mechanisms. We also briefly overview the role of statistical testing for PRNGs.

2.1 Kleptographic attacks

When kleptographic attacks were presented by Young and Yung [18], they were based on two concepts: black-box cryptography and asymmetric backdoors. The authors claim that the black-box model of cryptography is what allows for easy contamination, as users will be unaware of subversions in the affected devices. The asymmetric nature of the subversion provides an additional guarantee that if the attack is discovered, only the attacker will be able to make use of it.

The black-box assumption was somewhat relaxed in later years by other authors. The term kleptography has often been used in relation to the backdoored Dual EC DRBG, despite the backdoor existing in the specification itself rather than the implementation [16]. Some use the term kleptographic as synonymous with asymmetric in such cases [6]. Young and Yung have retained focus on black-box cryptography, however [21].

Algorithm substitution attacks (ASAs) are closely related and can be considered as a subset of kleptography [4]. However, many ASAs do not adhere to the asymmetry requirement [8]. This means that anyone who finds the subversion can exploit it, so it relies more strongly on the black-box assumption. This stems from the fact that a backdoored implementation that is neither black-box nor asymmetric is just a system vulnerability.

The first attack presented as kleptographic is SETUP (Secretly Embedded Trapdoor with Universal Protection). A SETUP is any modification to a cryptosystem whose input and output agree with the specification, and contains the attacker’s encryption but not decryption mechanism. It was demonstrated with the Pretty-Awful-Privacy (PAP) attack on RSA key generation [18] and Discrete Log attack against Diffie-Hellman [19].

Two attacks were presented as ASAs by Bellare, Paterson, and Rogaway [5]. The first are IV-replacement attacks on symmetric encryptions that surface their IVs. The second are more general biased-ciphertext attacks that compromise the associated data of some encryptions. The biased-ciphertext attack was later refined to be stateless and applicable to any randomized encryption scheme [4]. Input-triggered attacks have also been considered as particularly difficult to detect [8].

An important observation made in relation to successful subversions is the role of randomization in cryptography [5]. Sources of randomness are often used in encryption to approximate perfect secrecy [17], which demands that ciphertexts tell us nothing about the corresponding plaintexts. By allowing multiple valid ciphertexts for a plaintext, it becomes harder for an adversary to match them, strengthening cryptographic properties.

However, it is exactly this choice between multiple ciphertexts that weakens schemes in kleptographic scenarios. A subverted encryption scheme can in such cases be selective about which ciphertexts to output. If done properly, this can be both advantageous to an attacker and undetectable to others. It follows that unique ciphertext schemes, which allow only one valid ciphertext per plaintext, associated data, key and state, are preferable against kleptography [5].

2.2 Pseudo-random number generation

Sources of randomness are often required by consuming cryptographic applications for generation of session keys, nonces and similar values. Because it is expensive to observe natural random processes for this purpose, special generators are used instead [17]. These are most commonly called pseudo-random number generators (PRNGs) or deterministic random bit generators (DRBG). Informally, they operate by taking a short input string called a seed and extending it into a longer output that appears random.

Due to their extensive use, PRNGs have long been the target of malicious attacks, as outlined by Dodis et al. [9]. In 2015 Blomqvist presented a kleptographic PRNG construction based on the SETUP model, and also demonstrated its use in communication protocols, namely UMTS [6]. Faulty random number generation has been shown to compromise TLS and SSH servers [10], virtual machines [14], and the Linux kernel boot process [12].

In order for a PRNG to be fit for cryptographic use, it must satisfy certain properties. Informally speaking, its output must not only appear random, but also be difficult enough to predict. There are two important PRNG attributes defined by NIST in their Recommendation: backtracking resistance and prediction resistance [2]. PRNGs that possess these attributes are sometimes called cryptographically secure PRNGs (CSPRNGs).

Backtracking resistance is provided at time T if, given a PRNG's subsequent internal state, an attacker cannot distinguish past outputs from random output, and cannot recover past internal states. Prediction resistance is provided at time T if, given a PRNG's current state, an attacker cannot distinguish future outputs from random output, and cannot accurately predict future internal states. While all PRNG constructions in the NIST Recommendation provide backtracking resistance, prediction resistance is only guaranteed if enough entropy is available between consecutive generate requests.

When discussing backdoored encryption mechanisms, we quickly come across the decryptability condition [5]. Although randomized schemes provide subverted implementations some freedom of choice, they are still limited by having to produce a valid ciphertext. This is sometimes formalized by having the ciphertext be decryptable in two different ways: one by the intended recipient, and another by the attacker. Because PRNGs are not limited by decryptability, this restriction is lifted. Instead, the output must be decryptable only by the attacker's algorithm on top of being pseudo-random.

An interesting result along these lines, presented by Dodis et al. [9], is that a kleptographic PRNG construction is equivalent to a pseudo-random public key encryption scheme. In both cases we require generation of a public and secret key, an encapsulation algorithm that produces a pseudo-random output with the public key and additional information, and a decapsulation algorithm that recovers the additional information given the output and the private key. In a PRNG, the additional input is the secret internal state, while in public key encryption, it is a plaintext.

In order to confirm that a PRNG output appears random, statistical tests are sometimes used. For naïve applications, appearing random in this sense may be confined to specific aspects, e.g. having the same probability of producing 0s and 1s, but allowing sequences to be otherwise predictable. For cryptographic applications, outputs should appear random in every meaningful way, or an adversary will be able to correctly predict future outputs with non-negligible probability [3]. Statistical tests based on random processes can help eliminate some of these doubts.

Any kind of random process (e.g. numeric, geometric or game-related) can be used to produce a statistical test, so long as the correct outcome is known a priori. This outcome must be given in a probabilistic manner since randomness itself is probabilistic. As there are infinite patterns that could exist in a given sequence, the number of possible statistical tests is also infinite. Therefore we can never be fully assured that there is no bias in our PRNG, only check common points of failure. Two established batteries of tests are the Dieharder Suite [7] and the NIST Statistical Test Suite [3].

While potentially useful, statistical testing has its downsides. One is the aforementioned limitation of any finite battery, which cannot ensure randomness, only require it. Another is the misleading nature of the tests, as they operate on given output sequences, which are by definition known and entirely predictable. Even if the sequences do not display detectable patterns, the same cannot be said about past or future outputs. Additionally, their probabilistic nature means that even a truly random sequence will invariably fail some tests, and care must be taken when drawing any conclusions [3].

3 A KLEPTOGRAPHIC HASH PRNG

The NIST Special Publication 800-90A Revision 1, titled Recommendation for Random Number Generation Using Deterministic Random Bit Generators [2], and here referred to simply as the (NIST) Recommendation, contains specifications for three different PRNGs. These are the hash-based Hash DRBG and HMAC DRBG, and the CTR DRBG based on a block cipher algorithm. The controversial Dual EC DRBG was contained in this Recommendation at publication in 2012, and withdrawn with the currently valid Revision 1 in 2015.

The PRNGs described in the Recommendation were chosen due to the thorough nature of the specification, as well as their unique structure. The three constructions in the Recommendation share a common framework, a functional model that exposes the core functionality of the PRNG to consuming applications. These functions take care of many important administrative tasks and invoke more specific internal algorithms, which are what actually differ between the PRNG types. The Hash DRBG was arbitrarily selected among the three types for implementation.

A very important notion in the PRNG structure is that of the cryptographic mechanism boundary, as seen in Figure 1. The instantiate, reseed, generate and uninstantiate functions represent the functional model, and all except uninstantiate invoke an algorithm specific to the PRNG type. In order to retain important information in between function invocations, internal states with secret values and administrative information are stored by the PRNG. These must be protected and are not surfaced during operation, with reference values alone given to consuming applications.

The additional outward-facing function is the health check. PRNGs are mandated by this Recommendation to check that their instantiate, reseed and generate functions produce expected results using known-answer testing. As specific inputs cannot be fed into the outward-facing functions, it is assumed that these pertain to the internal algorithms specific to each type. Additionally, implementations must undergo validation testing for conformance by an accredited laboratory.

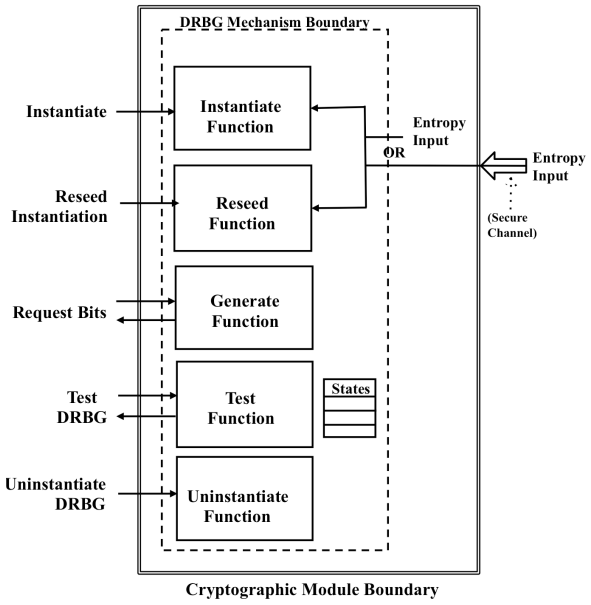

Figure 1. The PRNG mechanism functions. Taken from NIST Special Publication 800-90A (Revision 1, p. 16).

If a PRNG produces unexpected results during testing, it enters a critical error state and must not perform instantiate, reseed or generate operations. The critical error is conveyed through a catastrophic error flag, which is also returned if the entropy source fails. Less serious errors are conveyed using a regular error flag and may indicate a bad request or an unavailable resource. In other cases a success flag is output, and these status flags represent an important aspect of communication across the PRNG boundary.

The attack chosen for implementation is the algorithm substitution presented by Bellare, Jaeger and Kane [4], a stateless biased-ciphertext attack based on rejection sampling. The attack was designed for encryption mechanisms, where the plaintext is repeatedly encrypted with random side inputs until a suitable output is found. Each ciphertext is fed through a keyed pseudo-random function producing two values: the location and value of a single secret key bit. If the pair is incorrect, the process is repeated up to some number of times, then outputting the last result to avoid extreme slow-down.

It is a relatively weak attack, designed to function for any encryption scheme. It is strongly undetectable [4], which is not surprising since all outputs are generated by legitimate mechanisms. However, it does not meet the asymmetry standard of kleptography, as the same pseudo-random function is used during encryption and recovery of leaked bits. If someone can determine its key or otherwise gain access to it, they will also benefit from the attack. If this is easier than directly cracking the ciphertext, it weakens the encryption's security.

While asymmetric encryption can be used for PRNG output as shown by Dodis et al. [9], the mechanisms involved are slower than what is expected from a PRNG. Symmetric encryption is likely a better alternative than biased outputs for PRNG subversion, but this might require the attacker to observe specific outputs depending on the design, which is not guaranteed. Additionally, we expected the NIST DRBG design to be difficult to subvert, and a weaker attack can be used to prove stronger security, rather than the other way around.

However, an improvement to the attack's throughput was considered. Originally the attack produces a single bit out of a single output, with some chance of failure. Instead, we can attempt to extract multiple bits with a higher chance of failure. During PRN generation, a larger number of blocks can be generated, finally outputting the one hiding the most correct bits. The number of attempts and bits to leak is specified by the attacker, and the generation process is cut short if a block with all bits correct is found.

In order to validate our implementation of the Hash DRBG and its kleptographic variant, statistical testing was performed on sample outputs. Because they are based on the NIST Recommendation, the Statistical Test Suite (STS) by NIST [3] was used. The STS is provided as a software package with source code for fifteen distinct tests, which can be compiled for a target platform using the included makefile. It supports evaluation of PRNG output sequences via provided files or by extending the source code to directly feed output into the program. It also allows for inclusion of additional statistical tests by the user.

4 Implementation of the Attack

The legitimate Hash PRNG and its kleptographic subversion were coded in Python using the PyCryptodome module [1]. The implementation is split into three parts in order to better facilitate code structure. The base class DRBG contains the NIST functional framework, but lacks type-specific algorithms and known-answer testing. The class HashDRBG inherits the framework and implements the missing functionality. The KHashDRBG classes inherit HashDRBG and override the functions required to execute the attack. The code is available online in a public GitHub repository [13].

The DRBG states were implemented as separate classes, again split into a general and type-specific states. They contain administrative and secret information for DRBG instantiations, and are stored by the DRBG in a dictionary with integers as keys. These are then given to users as reference values. A special health state class is used to observe catastrophic failures that persist across instantiations, and execute known-answer testing on the generate algorithm at regular intervals. Status flags were also implemented as a separate enumeration class.

While the standard mandates the use of an approved entropy source, this was simplified. Instead, a simple os.urandom invocation is used to retrieve the minimum number of bits requested. The same is done when generating nonces, with bit-length equal to half of the required security strength. Known-answer testing was implemented by randomly generating ten sets of valid input parameters per available hash function for the instantiate, reseed and generate algorithms. These are stored together with expected results in a database using the sqlite3 module [11], and are retrieved every time health testing is requested.

Due to the layered structure of the DRBG, two vectors of attack were considered for subverting bit generation. The first overrides the HashDRBG-specific generate algorithm, and consequently also overrides its health testing so that it does not fail (though testing is still executed). The second overrides the public generate function, giving it more power to ignore prediction resistance requests and mandatory reseeds. However, it cannot avoid as much computational overhead as the type-specific attack. The two subversions are otherwise largely identical in their operation.

The attack uses HMAC-SHA512 as the pseudo-random function, to provide security at least as strong as the strongest hash function supported by the Hash DRBG. Its output is 512 bits long, which allows us to leak up to 46 secret bits, as each requires 10 bits to store its location and 1 bit to store its value. The attacker can specify the HMAC key, bits to leak per request, and additional generate attempts when instantiating the KHashDRBG class. For each generated block the number of correctly leaked bits is computed and the best block is stored.

In order to test the efficacy of our more complex biased output attack, which allows for multiple bits to be leaked at once, two measures were observed across parameter settings. As the leaked bit is never guaranteed to be correct so long as there is a specified maximum number of attempts, we measure the probability that the leaked bits are correct across 10.000 total generate requests from 100 different DRBG instantiations. From the observed probability it is trivial to compute the amount of missing information per bit.

This missing information then allows us to compute how much actual information we gain per leaked bit, which gives us the total amount of leaked information per generate request. The second measure used was how much the attack slowed down the generate process, averaged across 10.000 generate requests and compared to the legitimate implementation. The additional time needed was divided by the time of the legitimate implementation to display the relative slowdown per each parameter setting, and was computed for each vector of attack separately.

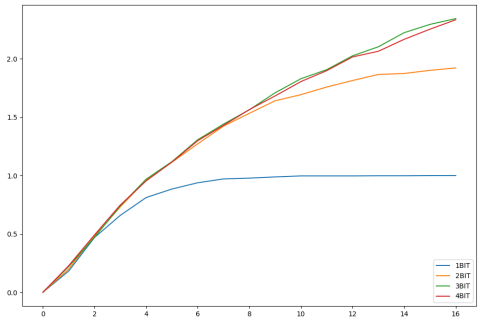

Figure 2. Bits of information gained by the attacker from a generate request (y axis) per number of additional generate attempts (x axis), for up to 4 leaked bits per request.

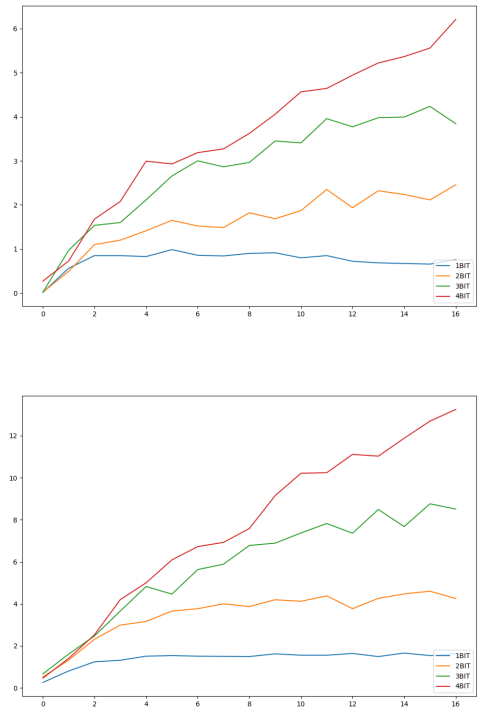

Figure 3. Relative additional time needed for generation (y axis) per number of additional generate attempts (x axis), for up to 4 leaked bits per request. Computed separately for HashDRBG-specific attack (top) and general attack (bottom).

Figure 2 displays the plots for total gained information. They follow similar trajectories and branch off at their limits, with the plot for one leaked bit plateauing under one bit of gained information. Interestingly, more leaked bits do not necessitate more gained information. Figure 3 displays additional time needed. Each leaked bit slows computation, with the general attack being around twice as slow as the HashDRBG-specific one.

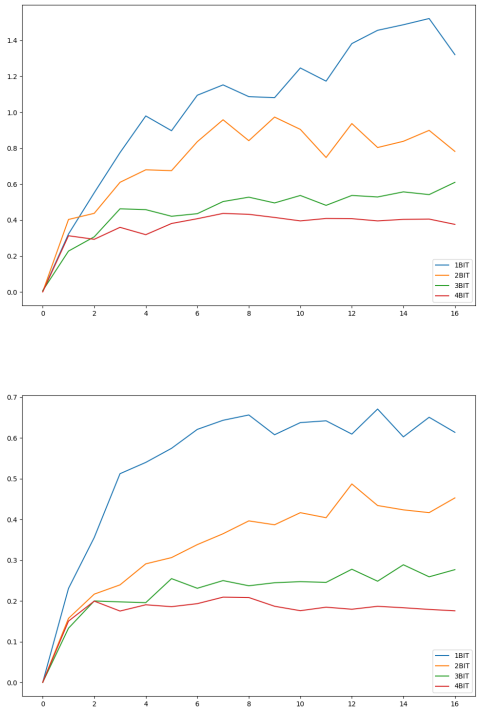

Figure 4. Ratio of gained information to additional time (y axis) per number of additional generate attempts (x axis), for up to 4 leaked bits per request. Computed separately for HashDRBG-specific attack (top) and general attack (bottom).

If we take the ratio between the two, we obtain a final measure of efficiency, seen in figure 4. This represents the total amount of information gained by the attacker from an output per additional time needed for generation, scaled to that of a legitimate implementation. It is clear that despite the potential to leak multiple bits, our implementation is not very effective, needing more time on average to leak the same amount of information. We therefore consider this adjustment to be unsuccessful. However, leaking two or more bits might still be preferred when only a certain number of outputs can be observed, as more total information is still leaked per request.

In order to verify our legitimate HashDRBG implementation, and that no excessive bias was introduced into the output by this attack, all three variants were tested using the NIST Statistical Test Suite (STS) with a significance rate of 0.01. Different settings were used for each of the subversions. For the general subversion, a single bit was leaked per request and up to 32 additional attempts were allowed, as the larger limit does not seriously affect its average speed. For the HashDRBG-specific subversion, two bits were leaked per request and up to 8 additional attempts were allowed, as it is somewhat more efficient.

A 1.000.000-bit output was investigated for each supported hash function, generated by conjoining outputs of 1024-bit requests. The suite was used with all 15 supported tests, and it executed some more than once. Cumulative Sums and Serial were executed twice, Random Excursions 8 times, Random Excursions Variant 18 times, and Non-Overlapping Template 148 times per input file. Some problems were encountered during the suite's use, possibly due to portability issues. Namely, the suite did not allow for input files to be parsed as multiple bitstreams, and p-values were not stored in the final results analysis.

Additionally, the Random Excursions and Random Excursions Variant failed for many input files due to an insufficient cycles error. As this parameter was hard-coded and could not be set during execution, the results were ignored. The tests failed on four hash functions for the legitimate DRBG (SHA-384, SHA3-512, SHA-512-224, SHA-512-256) and five hash functions for both of the subversions (SHA-224, SHA-384, SHA3-224, SHA3-256, SHA-512-256 for the general one and SHA-224, SHA-512, SHA3-256, SHA3-384, SHA3-512 for the HashDRBG-specific one).

Consequently, the combined number of tests for each output varied. For the legitimate implementation 1776 test were executed. The most successful output was from the SHA-512-224 function, where all 162 tests passed. The least successful was SHA3-512, where 4 out of 162 tests failed. In total, 20 tests failed across all hash functions, consisting of 16 Non-Overlapping Template, 2 Random Excursions, 1 Block Frequency and 1 Serial.

For both of the subverted implementations, 1750 tests were executed. For the general subversion, the SHA-224 function output passed all 162 tests, and the least successful was SHA-512-224, which failed 3 out of 162 tests. In total 12 tests failed, all Non-Overlapping Template. For the HashDRBG-specific subversion, SHA3-224 and SHA-512-256 outputs passed all 188 tests, and SHA-512 was least successful, failing 3 out of 162 tests. A total of 13 tests failed, consisting of 10 Non-Overlapping Template, 1 Universal, 1 Approximate Entropy and 1 Random Excursions Variant.

These results give us a 98.87% success rate for the legitimate implementation, which is close to expected for a significance rate of 0.01. Although some hash functions performed worse than others, all were sourced from the PyCryptodome module and had no effect on the structure of the DRBG. As such, we conclude that our implementation is correct. For the general subversion we have a 99.31% success rate, and for the HashDRBG-specific one a 99.26% success rate. In both cases the success rate was higher than for the legitimate implementation, so we conclude that our attack is not detectable through statistical testing.

5 Effectiveness of the attack

When evaluating the implemented attack in terms of practicality, it is important to recognize that there are many other was to subvert a PRNG that would leak information more readily. Aside from asymmetric encryption, which meets all necessary criteria except speed, symmetric encryption could also be used, although some form of randomization would have to prevent patterns in outputs. If the lack of entropy is not noticeable, a portion of secret bits could simply be fixed at every instantiation to a value known by the attacker.

The strength of our approach is in its undetectability. By continuing to use the legitimate bit generation algorithm to produce outputs, we do not introduce enough bias to be detected by the recommended statistical tests. This allows us to investigate how robust the NIST DRBG construction is against this type of subtle attack. Russell et al. [16] use the term watchdog to describe a program which attempts to detect a subversion. They define the offline watchdog, which interrogates provided implementations, and the online watchdog, which actively monitors the implementation and its outputs.

Even though the NIST DRBG standard mandates that the generate algorithm is tested at a specified interval, all internal known-answer testing falls strictly under the role of an offline watchdog. The procedure only checks the validity of the function's outputs for specified parameters, and does not monitor it in real time. If given the answers for a legitimate implementation, the HashDRBG-specific implementation will fail this testing. However, if the subversion happens on the level of the functional model or the testing is subverted as well, it will not be detected. Therefore, it is not a very effective defence.

Dodis et al. [9] suggest immunizing a backdoored PRNG through the use of a randomly seeded function on its output. However, they also point out that if the attacker gains access to this function, the subversion can still be successful. Randomizing our subverted output would indeed prevent the attacker from recovering information, but it is trivial to include the randomization in the generate procedure. Therefore, this extends protection only to consumers who secretly randomize the output before exposing it to others, and is not a good defence for any kind of standard.

A better use of randomization is suggested by Russell et al. in their split-program model [16]. They suggest resolving the problem of randomization in cryptographic schemes by separating all random parameter generation from the deterministic algorithms that consume them. In our case, it could mean separating instantiating PRNG states from bit generation, or separating the entropy source used by the DRBG from its consequent use. This is in a way already built into the construction as the NIST functional model.

The authors use this model to create a subversion-resistant PRNG construction. By re-randomizing the random parameter with some function before feeding it to the deterministic generate algorithm, any embedded trapdoor is destroyed. However, this hinges on the fact that the generate algorithm cannot deviate from expected outputs by randomizing its outputs. In our implementation we found it was enough to simply increment the parameters without randomizing them, producing a deterministic output given a set of inputs.

While this output is different from what is expected from a legitimate implementation, testing was also shown to be easy to subvert. If testing cannot be subverted, then requiring the output of the generate algorithm to be deterministic is not enough, as we can still embed information into the output. This also allows us to subvert known-answer testing by simply changing the set of answers to what is consistent with our implementation. However, if the secret state is randomized, this is harder to overcome.

If the secret state of the PRNG is randomized before it is fed to the generate algorithm, our implementation will not be able to leak information. This can be resolved by attempting to recover the true secret state from another source, which should not be too difficult. If the secret state is overwritten by its randomization, the original value could still be stored somewhere by the subversion, and used by the attacker to recover consequent randomizations. The only effective way to prevent the attack is therefore randomization using fresh entropy, which is implemented as prediction resistance.

Prediction resistance for bit generation mandates that the secret state is reseeded using fresh entropy before generating any bits. While the entropy used for this process could be subverted, this was not considered in our work, as the NIST standard mandates the use of approved sources whose failure can be detected. The general vector of attack we implemented does ignore the reseed, but this might be detected during validation testing, as the subversion produces the same output with or without prediction resistance for a given state.

On the one hand, this does give us a different way of subverting known-answer testing. Since leaking bits with prediction resistance is not possible, and testing mandates the use of prediction resistance if implemented, the generate algorithm would simply use the legitimate procedure when called with this parameter. On the other hand, it effectively guards against our attack, as every request with prediction resistance would destroy the attacker's knowledge of any bits of the current secret state. As prediction resistance is only guaranteed with this parameter, we conclude that the NIST construction is safe through its use.

6 Conclusions

Initially defined by Young and Yung, kleptography has proliferated over the years and has been applied to symmetric encryption [5], key generation protocols [19], and PRNGs [9] among others. A commonly cited example of kleptography is the Dual EC DRBG, even though the backdoor exists in the specification and does not use the black-box model. Closely related are algorithm substitution attacks (ASAs), which often drop the asymmetry requirement instead, and are thus more reliant on the black-box assumption.

Pseudo-random number generators (PRNGs), also called deterministic random bit generators (DRBGs), are commonly used by cryptographic protocols as an efficient supply of random values. In order for a PRNG to be cryptographically secure, the NIST Recommendation [2] highlights two key properties: backtracking resistance and prediction resistance. PRNGs are particularly interesting for kleptographic attacks because their output does not have to meet the decryptability condition, and kleptographic PRNGs have been shown to be equivalent to pseudo-random public key cryptography [9].

The National Institute of Standards and Technology (NIST) has issued a recommendation for building cryptographically secure PRNGs [2]. These rely on a general, public function framework, with the detailed operations of each function implemented using type-specific algorithms. The current standard supports the two hash-based Hash DRBG and HMAC DRBG, and the CTR DRBG based on a block cipher, and also mandates the use of internal known-answer testing. The Hash DRBG was arbitrarily chosen amongst these and was implemented in Python.

The implementation was then subverted using a variant of the biased-ciphertext attack [4], generating multiple outputs until one a suitable one is found. This output reveals a secret state bit when fed through a keyed pseudo-random function. It is a relatively weak attack, but also highly undetectable, since all of its outputs are generated by legitimate algorithms. We attempted to improve the attack by allowing it to leak multiple bits, but this was found to be relatively inefficient when considering the additional time needed for the subverted generation.

Both the legitimate and kleptographic implementations were tested using the NIST Statistical Test Suite [3], which contains 15 statistical tests to check for common biases in PRNGs. A 1.000.000-bit output was generated for each of the supported hash functions. For the default significance rate of 0.01, the legitimate implementation passed 98.87% of tests, the kleptographic subversion of the functional model passed 99.31% of tests, and the kleptographic subversion of the generate algorithm passed 99.26%, supporting our claim that the implementations are correct.

When evaluating the effectiveness of the attack, the concept of a watchdog is convenient [16]. The known-answer testing mandated by NIST is a relatively weak form of the offline watchdog, as it can be subverted as easily as the generate algorithm. If testing cannot be subverted, then we can also attempt to replace the provided output values, as our subversion is deterministic. Because testing demands the use of prediction resistance if it is available, we can also avoid detection by only leaking bits when prediction resistance is not requested.

Since prediction resistance demands that the internal secret state is reseeded with fresh entropy, this effectively guards against our attack, with every reseed destroying all currently known secret bits. This represents a strong form of parameter randomization proposed by Russell et al. [16]. Although prediction resistance requests could be ignored by the subversion, this might be detectable during validation testing. As prediction resistance is only guaranteed for the NIST DRBG with this request, we conclude that the NIST DRBG construction is safe against our attack.

In the future, symmetric encryption could be used to leak a larger amount of information, replacing the PRNG output with ciphertext decryptable by the attacker. The provided entropy during instantiation and reseeds could instead be used to randomize the encryption process and ensure that the output passes statistical checks for pseudo-randomness. Subversion of the entropy generation process could also be investigated further. We hope that the field of kleptography and algorithm subversion continues to evolve and helps expose weak points of cryptographic protocols.

References

[1] *Pycryptodome*. https://www.pycryptodome.org, Accessed: 2022-08-14.

[2] Barker, Elaine B, John Michael Kelsey, *et al.*: *Recommendation for random number generation using deterministic random bit generators (revised)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2007.

[3] Bassham III, Lawrence E, Andrew L Rukhin, Juan Soto, James R Nechvatal, Miles E Smid, Elaine B Barker, Stefan D Leigh, Mark Levenson, Mark Vangel, David L Banks, *et al.*: *Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications*. National Institute of Standards & Technology, 2010.

[4] Bellare, Mihir, Joseph Jaeger, and Daniel Kane: *Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks*. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1431–1440, 2015.

[5] Bellare, Mihir, Kenneth G Paterson, and Phillip Rogaway: *Security of symmetric encryption against mass surveillance*. In *Annual Cryptology Conference*, pages 1–19. Springer, 2014.

[6] Blomqvist, Ferdinand *et al.*: *Kleptography–overview and a new proof of concept*. Master's thesis, 2015.

[7] Brown, Robert G, Dirk Eddelbuettel, and David Bauer: *Dieharder: A random number test suite*. https://webhome.phy.duke.edu/∼rgb/General/dieharder.php, Accessed: 2022-08-11.

[8] Degabriele, Jean Paul, Pooya Farshim, and Bertram Poettering: *A more cautious approach to security against mass surveillance*. In *International Workshop on Fast Software Encryption*, pages 579–598. Springer, 2015.

[9] Dodis, Yevgeniy, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart: *A formal treatment of backdoored pseudorandom generators*. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 101–126. Springer, 2015.

[10] Heninger, Nadia, Zakir Durumeric, Eric Wustrow, and J Alex Halderman: *Mining your ps and qs: Detection of widespread weak keys in network devices*. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220, 2012.

[11] Hipp, Richard D and Gerhard Häring: *sqlite3*. https://docs.python.org/3/library/sqlite3.html, Accessed: 2022-08-14.

[12] Mowery, Keaton, Michael Wei, David Kohlbrenner, Hovav Shacham, and Steven Swanson: *Welcome to the entropics: Boot-time entropy in embedded devices*. In *2013 IEEE Symposium on Security and Privacy*, pages 589–603. IEEE, 2013.

[13] Rajh, Mihael: *Kleptoprng*. https://github.com/mikethenut/KleptoPRNG, Accessed: 2022-08-14.

[14] Ristenpart, Thomas and Scott Yilek: *When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography.* In *NDSS*, 2010.

[15] Roumani, Yaman: *Detection time of data breaches*. Computers & Security, 112:102508, 2022.

[16] Russell, Alexander, Qiang Tang, Moti Yung, and Hong Sheng Zhou: *Cliptography: Clipping the power of kleptographic attacks*. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 34–64. Springer, 2016.

[17] Stinson, Douglas R: *Cryptography: theory and practice*. Chapman and Hall/CRC, 2005.

[18] Young, Adam and Moti Yung: *The dark side of "black-box" cryptography or: Should we trust capstone?* In *Annual International Cryptology Conference*, pages 89–103. Springer, 1996.

[19] Young, Adam and Moti Yung: *Kleptography: Using cryptography against cryptography*. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 62–74. Springer, 1997.

[20] Young, Adam and Moti Yung: *Malicious cryptography: Kleptographic aspects*. In *Cryptographers' Track at the RSA Conference*, pages 7–18. Springer, 2005.

[21] Young, Adam and Moti Yung: *Kleptography from standard assumptions and applications*. In *International Conference on Security and Cryptography for Networks*, pages 271–290. Springer, 2010.

[22] Young, Adam L and Moti M Yung: *Space-efficient kleptography without random oracles*. In *International Workshop on Information Hiding*, pages 112–129. Springer, 2007.