Cryptanalysis of the Random Number Generator of the Windows Operating System

LEO DORRENDORF and ZVI GUTTERMAN
The Hebrew University of Jerusalem
and
BENNY PINKAS
University of Haifa

The PseudoRandom Number Generator (PRNG) used by the Windows operating system is the most commonly used PRNG. The pseudorandomness of the output of this generator is crucial for the security of almost any application running in Windows. Nevertheless, its exact algorithm was never published.

We examined the binary code of a distribution of Windows 2000. This investigation was done without any help from Microsoft. We reconstructed the algorithm used by the pseudorandom number generator (namely, the function `CryptGenRandom`). We analyzed the security of the algorithm and found a nontrivial attack: Given the internal state of the generator, the previous state can be computed in 2^{23} steps. This attack on forward security demonstrates that the design of the generator is flawed, since it is well known how to prevent such attacks. After our analysis was published, Microsoft acknowledged that Windows XP is vulnerable to the same attack.

We also analyzed the way in which the generator is used by the operating system and found that it amplifies the effect of the attack: The generator is run in user mode rather than in kernel mode; therefore, it is easy to access its state even without administrator privileges. The initial values of part of the state of the generator are not set explicitly, but rather are defined by whatever values are present on the stack when the generator is called. Furthermore, each process runs a different copy of the generator, and the state of the generator is refreshed with system-generated entropy only after generating 128KB of output for the process running it. The result of combining this observation with our attack is that learning a single state may reveal 128KB of the past and future output of the generator.

The implication of these findings is that a buffer overflow attack or a similar attack can be used to learn a single state of the generator, which can then be used to predict all random values, such as SSL keys, used by a process in all its past and future operations. This attack is more severe and more efficient than known attacks in which an attacker can only learn SSL keys if it is controlling the attacked machine at the time the keys are used.

Categories and Subject Descriptors: D.4.6 [**Security and Protection**]:

General Terms: Security

The work of B. Pinkas supported in part by a grant from the Israel Science Foundation.
Authors' addresses: L. Dorrendorf and Z. Gutterman, The Hebrew University of Jerusalem; email: {dorrel, zvikag}@cs.huji.ac.il; B. Pinkas, University of Haifa; email: benny@pinkas.net.

DOI 10.1145/1609956.1609966 http://doi.acm.org/10.1145/1609956.1609966

Additional Key Words and Phrases: Pseudorandom number generator (PRNG), Windows operating system, cryptanalysis

ACM Reference Format:

Dorrendorf, L., Gutterman, Z., Pinkas, B. 2009. Cryptanalysis of the random number generator of the windows operating system. ACM Trans. Info. Syst. Sec. 13, 1, Article 10 (October 2009), 32 pages. DOI = 10.1145/1609956.1609966 http://doi.acm.org/10.1145/1609956.1609966

1. INTRODUCTION

Almost all cryptographic systems are based on the use of a source of random bits, whose output is used, for example, to choose cryptographic keys or choose random nonces. The security analysis (and proofs of security) of secure systems are almost always based on the assumption that the system uses some random data (e.g., a key), which is uniformly distributed and unknown to an attacker. The use of weak random values may result in an adversary being able to break the system (e.g., weak randomness may enable the adversary to learn the cryptographic keys used by the system). This was demonstrated, for example, by the analysis of the implementation of SSL in Netscape [Goldberg and Wagner 1996] or in an attack predicting Java session-ids [Gutterman and Malkhi 2005].

Generation of pseudorandom numbers. Physical sources of randomness are often too costly; therefore, most systems use a pseudoandom number generator. The generator is modeled as a function whose input is a short random seed and whose output is a long stream, which is indistinguishable from truly random bits. Implementations of pseudorandom generators often use a state whose initial value is the random seed. The state is updated by an algorithm, which changes the state and outputs pseudorandom bits and implements a deterministic function of the state of the generator. The theoretical analysis of pseudorandom generators assumes that the state is initialized with a truly random seed. Implementations of pseudorandom generators initialize the state with random bits ("entropy"), which are gathered from physical sources, such as timing of disk operations, of system events, or of a human interface. Many implementations also refresh (or "rekey") the state periodically by replacing the existing state with one that is a function of the existing state and of entropy similar to that used in the initialization.

Security properties. A pseudorandom number generator must be secure against external and internal attacks. An attacker might know the algorithm (or code) that defines the generator, know the output of the generator, be able at some point to examine the generators's state, and have partial knowledge of the entropy used for refreshing the state. We list here the most basic security requirements that must be provided by pseudorandom generators, using common terminology (e.g., as used in Barak and Halevi [2005]).

—*Pseudorandomness.* The generator's output looks random to an outside observer.

—*Forward security.* An adversary that learns the internal state of the generator at a specific time cannot learn anything about previous outputs of the generator.

—*Backward security* (also known as *break-in recovery*). An adversary that learns the state of the generator at a specific time does not learn anything about future outputs of the generator.
Note that the generator operates as a deterministic process and, therefore, knowledge of the state at a specific time can be used to compute subsequent states and outputs (by simply simulating the operation of the algorithm run by the generator). Consequently, there is no way to ensure backward security unless the state of the generator is refreshed with external data ("entropy") that is sufficiently random. This process must be done quite often in order to ensure backward security.[1]

Forward security is concerned with ensuring that the state of the generator does not leak information about previous states and outputs. If a generator does not provide forward security, then an attacker that learns the state at a certain time can learn previous outputs of the generator and, consequently, past transactions of the user of the system. (Consider, for example, an attacker that targets a computer in an Internet cafe and learns keys used by previous users of the machine. Another option for an attacker is to decide which machine to attack only after observing which machines are interesting as targets (e.g., machines that were used by a specific user or were used for specific transactions.) Forward security can be easily guaranteed by ensuring that the function that advances the state is one way. It is well-known how to construct forward-secure generators (for an early usage of such generators see, Beaver and Haber [1992]; see also Bellare and Yee [2003] for a comprehensive discussion and a generic transformation of any standard generator to one that provides forward security). Forward security is also a mandatory requirement of the German evaluation guidance for pseudorandom number generators (AIS 20) [Bundesamt für Sicherheit in der Informationstechnik 1999]. The fact that the random number generator used by Windows 2000 does not provide forward security demonstrates that the design of the generator is flawed.

The random number generator used by Windows. This article studies the pseudorandom number generator used in Microsoft Windows systems, which we denote as the WRNG. The WRNG is the most frequently used pseudorandom number generator, with hundreds of millions of instances running at any given time. It is used by calling the function `CryptGenRandom`. According to the book *Writing Secure Code* [Howard and LeBlanc 2002], published by Microsoft, the WRNG was first introduced in Windows 95 and was since embedded in all Windows-based operating systems, such as Windows XP and Windows 2000,

[1]One might wonder why the state of the generator should not be continuously updated with entropy gathered from system events. The reason for this is that in most cases entropy can only be gathered at a relatively slow rate, and it is preferable to gather sufficient amounts of entropy before refreshing the generator. See Barak and Halevi [2005] for a discussion of this subject.

and in all their variants.[2] According to Howard and LeBlanc [2002], the design of the WRNG has not changed between the different versions of the operating systems.

In this work, we examined the generator that is implemented in the Windows 2000 operating system (Service Pack 4). Windows 2000 is still a relatively popular operating system, especially in enterprises, with a market share of about 1.6% as of January 2009.[3] In Section 6.1, we describe our initial analysis of Windows XP and note that Microsoft acknowledged that Windows XP suffers from the vulnerabilities we identified in Windows 2000.

WRNG usage. The WRNG is used by calling the Windows system function `CryptGenRandom` with the parameters `Buffer` and `Len`. Programs call the function with the required length of the pseudorandom data that they need and receive as output a buffer with this amount of random data. The function is used by internal operating system applications such as the generation of TCP sequence numbers, by operating system applications, such as the Internet Explorer browser, and by applications written by independent developers.

Our contributions. This article describes the following results.

—We present a detailed analysis of the Windows pseudorandom number generator. The analysis is accompanied by a concise pseudocode for the entire implementation of the WRNG (the complete pseudocode is about 1,000 lines of code) and by a user-mode simulator of the WRNG. The analysis is based on examination of the binary code of the operating system; see details later in the text.

—We present an attack on the forward security of the WRNG. We show how an adversary can compute past outputs and states from a given state of the WRNG, with an overhead of 2^{23} computation (namely, in a matter of seconds on a home computer).

—We analyze the way in which the operating system uses the WRNG and note that a different copy of the WRNG is run, in user mode, for every process, and that typical invocations of the WRNG are seldom refreshed with additional entropy. Therefore, the WRNG does not provide a reasonable level of backward security. Furthermore, the attack on forward security, which only works while there is no entropy-based rekeying, is highly effective. In addition to these observations, we found out that part of the state of the generator is initialized with values that are rather predictable.

Attack model. Our results suggest the following attack model: The attacker must obtain the state of the generator at a certain time. This can be done by attacking a specific application and obtaining the state of the WRNG run by this process or by launching a buffer overflow attack or a similar attack providing

[2]The statement in Howard and LeBlanc [2002] was written before Windows Vista was released. The documentation of `CryptGenRandom` states that it is supported by Windows Vista, but we have not verified this statement.
[3]See, e.g., `http://www.w3schools.com/browsers/browsers_os.asp`.

administrator privileges and obtaining the state of the generators run by all processes. After learning the state, the attacker does not need any additional information from the attacked system. It can learn all previous and future outputs of the generator (from the last rekey to the next rekey) and subsequently learn cryptographic keys, such as SSL keys, used by the attacked system. Our attack is more powerful and more efficient than attacks that require the attacker to control the target machine at the time it is generating cryptographic keys, observe these keys, and relay them to the attacker (that type of attack is representative of adversaries that use operating system or program vulnerabilities, such as buffer overflows, to gain access to remote machines, or of attacks that are based on physical access, e.g., for the purpose of installing key loggers on specific machines).

Gap between theory and practice. The generation of pseudorandom numbers is a well-studied issue in cryptography, see, for example, Shamir [1981], and Blum et al. [1983]. One might therefore, be surprised to learn that constructing an actual implementation of a pseudorandom number generator is quite complex. There are many reasons for this gap between theory and practice.

—*Real-world attacks.* Actual implementations are prone to many attacks that do not exist in the clean cryptographic formulation that is used to design and analyze pseudorandom generators (consider, e.g., timing attacks and other side-channel attacks).
—*Seeding and reseeding the generator.* Generators are secure as long as they are initialized with a truly random seed. Finding such a seed is not simple. Furthermore, the state of the generator must be periodically refreshed with a fresh random seed in order to prevent backward security attacks. The developer of a generator must therefore, identify and use random sources with sufficient entropy.
—*Performance.* Provably secure generators might incur high-computation overhead. Therefore, even a simple PRNG, such as the Blum-Blum-Shub generator [Blum et al. 1983], is rarely used in practice.
—*Lack of knowledge.* In many cases, the developers of the system do not have sufficient knowledge to use contemporary results in cryptography.

These factors demonstrate the importance of having a secure pseudorandom generator provided by the operating system.[4] The designers of the operating system can be expected to be versed with the required knowledge in cryptography and know how to extract random system data to seed the generator. They can therefore, implement an efficient and secure generator. Unfortunately, our work shows that the Windows pseudorandom generator has several unnecessary flaws.

[4]Indeed, given the understanding that writing good cryptographic functions is hard, operating systems tend to provide more and more cryptographic functionality as part of the operating system itself. For example, Linux provides implementations of hash functions as part of its kernel.

1.1 Related Work

Existing PRNG implementations. In the past, PRNGs, were either a separate program or a standard library within a programming language. The evolution of software engineering and operating systems introduced PRNGs, which are part of the operating system. From a cryptographic point of view, this architecture is advantageous since it supports the initialization of the PRNG with operating system data (which has more entropy and is hidden from users).

Implementations of PRNGs can be either blocking or nonblocking. A blocking implementation does not provide output until it collects sufficient amount of system-based entropy. A nonblocking application is always willing to provide output. The PRNG of the FreeBSD operating system is described in Murray [2002]. FreeBSD implements a single nonblocking device, and the authors declare their preference of performance over security. The PRNG used in OpenBSD is described in de Raadt et al. [1999], which also includes an overview of the use of cryptography in this operating system. Castejon-Amenedo et al. [2003] propose a PRNG for UNIX environments. Their system is composed of an entropy daemon and a buffer manager that handles two devices—blocking and nonblocking. The buffer manager divides entropy equally between the two devices, such that there is no entropy that is used in both. A notable advantage of this scheme is the absolute separation between blocking and nonblocking devices, which prevents launching a denial-of-service attack on the blocking device by using the nonblocking device (such an attack is possible in Linux, as is shown in Gutterman et al. [2006]).

The Linux PRNG. The Linux operating system includes an internal entropy-based PRNG named `/dev/random` [Ts'o 1994]. Following Gutterman et al. [2006], we denote this generator as the LRNG. The exact algorithm used by the LRNG (rather than the source code, which is open) was published in Gutterman et al. [2006], where several security weaknesses of this generator were also presented. We discuss in detail in Section 6.2 the differences between the LRNG and the WRNG. We note here that the attack on the WRNG is more efficient, and that in addition, unlike the LRNG, the WRNG refreshes its state very rarely and is therefore, much more susceptible to attacks on its forward and backward security. On the other, the WRNG is not susceptible to denial of service attacks, which do affect the LRNG.

Analysis of PRNGs. A comprehensive discussion of the system aspects of PRNGs, as well as a guide to designing and implementing a PRNG without the use of special hardware or access to privileged system services, is given by Gutmann [1998]. Issues related to operating system entropy sources were discussed in a 2004 NIST workshop on random number generation [Kelsey 2004; Gutmann 2004]. An extensive discussion of PRNGs, which includes an analysis of several possible attacks and their relevance to real-world PRNGs, is given by Kelsey et al. [1998]. Additional discussion of PRNGs, as well as new PRNG designs appear in Kelsey et al. [1999] and Ferguson and Schneier [2003].

Barak and Halevi [2005] present a rigorous definition and an analysis of the security of PRNGs, as well as a simple PRNG construction. That work suggests separating the entropy extraction process, which is information-theoretic in nature, from the output generation process. Their construction is based on a cryptographic pseudorandom generator G, which can be implemented, for example, using AES in counter mode, and which does not use any input except for its seed. The state of the PRNG is the seed of G. Periodically, an entropy extractor uses system events as an input from which it extracts random bits. The output of the extractor is xored into the current state of G. The construction is much simpler than most existing PRNG constructions, yet its security was proved in Barak and Halevi [2005] assuming that the underlying building blocks are secure. We note that our analysis shows that the WRNG construction, which is much more complex than the work of Barak and Halevi [2005] (but also well predates it), suffers from weaknesses, which could have been avoided by using the latter construction.

Outline. The rest of the article is organized as follows. Section 2 provides a detailed description of the WRNG. Section 3 presents cryptanalytic attacks on the generator. Section 4 describes the interaction between the operating system and the generator and its security implications. Section 5 describes an analysis of the update process of one of the main variables used by the generator. Section 6 compares the WRNG to other generators (in particular, to the generator used by Linux), and Section 7 contains conclusions and suggestions for further research.

2. THE STRUCTURE OF THE WINDOWS RANDOM NUMBER GENERATOR

We start by discussing the way in which the function CryptGenRandom is used, followed by a description of the process we used to analyze the binary code. We then describe the main loop of the generator, the functions called by this loop, and the initialization of the generator. We conclude this section by describing the scope in which the generator is run by the operating system.

2.1 Using CryptGenRandom

In Linux, the output of the random-number generator can be read like any other device, and a byte stream is acquired directly from this device using a pipe or redirection. In the Windows operating system, the output of the RNG must be called through the system, function CryptGenRandom, which is part of the Windows Crypto API. The function does not receive a user supplied seed to reproduce a sequence of outputs, and is supposed to always generate unpredictable random data. Its output is intended for cryptographic and security purposes such as initial values (IVs), salts, and possibly sequence numbers. Cryptographic keys can be generated directly (as buffers of random data) using CryptGenRandom. (Another Crypto API function named CryptGenKey is used to generate specific types of keys used in the Crypto API algorithms, but internally it uses the same RNG as CryptGenRandom.)

Declaration. The function CryptGenRandom receives three parameters: an output buffer, its length, and a CSP handle (which we explain in the following text). It is declared in the following way:

```
BOOL WINAPI CryptGenRandom(
  HCRYPTPROV hProv,
  DWORD dwLen,
  BYTE* pbBuffer
);
```

A small example program of the usage of CryptGenRandom is shown in the following text.

```
#include <windows.h>
#include <wincrypt.h>
#define SIZE 160
void main() {
         HCRYPTPROV hProv = 0;
         BYTE data[SIZE];
         CryptAcquireContext(&hProv, NULL, NULL,
                             PROV_RSA_FULL, 0);
         CryptGenRandom(hProv, SIZE, data);
}
```

The function `CryptAcquireContext` in the code above chooses and initializes a cryptographic service provider (CSP), a DLL providing cryptographic functions. A CSP is a software library implementing a set of security algorithms, and its choice determines the implementation of the RNG. This work examines the Microsoft Enhanced Cryptographic Provider, the default CSP for strong cryptography, which is used by Internet Explorer (see `http://msdn2.microsoft.com/en-gb/library/aa386986.aspx`). The control flow of CryptGenRandom starts in `ADVAPI32.DLL`, which dispatches the call to the CPGenRandom function (CP stands for Crypto Provider), in the chosen CSP library. This library is `RSAENH.DLL` for the CSP we examined (Microsoft Enhanced Cryptographic Provider).

Another relevant binary is `KSECDD.SYS`, which is implementing the kernel part of the RNG. In the rest of this work, the name CryptGenRandom refers to all of the code involved. See Section 2.2 for a list of exact versions of the binaries we examined.

2.2 Analyzing the Binary Code

The algorithm employed by the WRNG, and its design goals, were never published. There are some published hints about the inner structure of the WRNG [Howard and LeBlanc 2002]. However, the exact design and security properties were not published.

We examined the Windows 2000 operating system. Windows 2000 was chosen by us, since its random number generator was relatively easy to examine, while sharing the majority of its design with the later Windows XP version. (The fact that Windows XP suffers from the vulnerabilities we identified in Windows 2000 was acknowledged by Microsoft, see Section 6.1.) Windows Vista was unreleased at the time this work was done.

Our entire research was conducted on the binary version supplied with each running Windows system. We did not have access to the source code of the generator. The research was conducted on Windows 2000 Service Pack 4 (with the following DLL and driver versions: `ADVAPI32.DLL` 5.0.2195.6876, `RSAENH.DLL` 5.0.2195.6611, and `KSECDD.SYS` 5.0.2195.824). The entire inspected binary code is over 10, 000 lines of assembly code.

Our study required static and dynamic analysis of the binary code. Static analysis is the process where the binary assembly code is manually translated into pseudocode written in a high-level programming language. In the dynamic analysis phase, the binary is run while a debugger is tracing the actual commands that are run, and the values of memory variables. The combined process of dynamic and static analysis enables us to focus on relevant functions only and better understand the meaning of variables and functions.

We used several tools in our analysis: the interactive disassembler (IDA) tool [Guilfanov 2006], which is an editor for static code analysis, the OllyDbg tool [Yuschuk 2004] for dynamic study of our user mode runtime environment, and the WinDBG tool [Microsoft 2006] as our kernel debugging tool. (See also the book *Reversing* [Eilam 2005], which provides an excellent introduction to the field of code analysis.) To verify our findings and demonstrate our attacks, we developed four tools:

—`CaptureCryptGenRandom`, which captures the current WRNG state into a file;
—`NextCryptGenOutputs`, which calculates future outputs of the WRNG from a given state;
—`PreviousCryptGenOutputs0`, which calculates previous outputs and states of the WRNG from a given state and knowledge of the initial State and R variables (this attack and the roles of State and R are described in Section 3.2);
—`PreviousCryptGenOutputs23`, which calculates previous outputs and states of the WRNG from a given state alone. (This attack in described in Section 3.2. It has an overhead of 2^{23} steps.)

These tools validate our findings. We currently do not publish the tools online. They can be provided upon request.

Pseudocode and function names. In this paper, we describe pseudocode, which is derived from disassembling the binaries of CryptGenRandom. This is not an exact translation to a higher language. Rather, the pseudocode is abridged and restructured for clarity: Functions are inlined or extracted to better convey the logical structure, and unused control paths, error handling, thread synchronization, exceptions, and checks for extreme cases are not shown. Nevertheless, the pseudocode is close enough to the original code to convey the algorithm.

The names of exported functions (such as CryptGenRandom or CPGenRandom) are preserved in compilation, but the names of internal functions are normally stripped and lost along with variable names. In this case, it is in general impossible to retrieve the original names of functions. However, some of the internal function names (such as `NewGenRandom` and `FIPS186Gen`

```
CryptGenRandom(Buffer, Len)
// output Len bytes to buffer
   while (Len>0) {
      R := R ⊕ get_next_20_rc4_bytes()
      State := State ⊕ R
      T := SHA-1'(State)
      Buffer := Buffer | T    // | denotes concatenation.
        //  See comment below for the case Len<20.
      R[0..4] := T[0..4]
        // copy 5 least significant bytes
      State := State + R + 1
      Len := Len − 20
   }
```

Fig. 1. The main loop of the WRNG. It has input parameters *Len*, which is the number of bytes to be output, and *Buffer*, which gets the output. All internal variables are 20 bytes long and uninitialized. *Buffer* is assumed to be empty and the WRNG output is concatenated to it in each round of the loop. The function SHA-1' is a variant of SHA-1 where the initialization vector (IV) is ordered differently. If $Len < 20$, then only the *Len* leftmost bytes of *T* are added to *Buffer*.

mentioned later in the text) could be restored by loading official symbol files, which are provided by Microsoft for debugging purposes.

2.3 The Structure of the Generator

The algorithm used by the generator is based on two common cryptographic primitives, the RC4 stream cipher (described in Appendix A) and the SHA-1' hash function, which maps arbitrary inputs to a 20-byte long output.

2.3.1 *The Code of the Generator.*

The main loop of the WRNG. The main loop, presented in Figure 1, generates 20 bytes of output in each iteration. The main state of the WRNG is composed of two registers, R and State, which are updated in each iteration and are used to calculate the output. The loop operates on data in chunks of 20 bytes: Each of the registers used in the main loop, R, State, and T, is 20 bytes long. This is also the length of the result of the internal function call get_next_20_rc4_bytes and of the output of SHA-1'. The output is generated in increments of 20 bytes.

The main loop uses the two variables, R and State, to store a state. It calls an internal function get_next_20_rc4_bytes to obtain 20 bytes of pseudorandom data and uses them to update R and State. It generates 20 bytes of output by applying SHA-1', a variant of SHA-1, to State and then updates State again using part of this output and using R. (The only difference between SHA-1'and the standard implementation of SHA-1 is that the byte order of the initial state is reversed, as is done, for example, in the usage of SHA-1 in the FIPS 180-2 standard.)

```
get_next_20_rc4_bytes()
{
   // if |output of RC4 stream| >= 16Kbytes then
   // refresh state
   while (RC4[i].accumulator >= 16384) {
       RC4[i].rekey();    // refresh with system entropy
       RC4[i].accumulator = 0;
       i = (i+1) % 8;
   }
   result = RC4[i].prng_output(20);
     // 20 byte output from i'th instance
   RC4[i].accumulator += 20;
   i = (i+1) % 8;
   return(result);
}
```

Fig. 2. Function `get_next_20_rc4_bytes()`.

The function get_next_20_rc4_bytes. As described earlier, the function `get_next_20_rc4_bytes` is used to refresh the state of the main loop of the generator. (This function is called `NewGenRandom` in Windows 2000 and resides inside `RSAENH.DLL`. We note that Windows XP uses for this purpose a different function, `SystemFunction036` which is exported from `ADVAPI32.DLL`. We instead denote the function by the name `get_next_20_rc4_bytes`, which describes the functionality of the function more clearly.) The function keeps a state that is composed of eight instances of the RC4 stream cipher. (See Appendix A for a description of RC4.) In each call, the function selects one RC4 state in a round-robin fashion, uses it to generate 20 bytes of output, and returns them to its caller. In the next call, it uses the next RC4 stream. After an RC4 instance generates 16KB of output, it is refreshed with entropy gathered from the system, as is described later in the text.

The function is described in Figure 2 (this description assumes a static variable `i`, which is initialized to zero before the first call). We can also imagine this function as storing eight output streams from eight independent invocations of RC4. The function holds a pointer `i`, which points to one of the streams, and for each stream (numbered i), it holds a counter c_i that points to a location in the stream (in the code of Figure 2 this counter is denoted by `RC4[i].accumulator`). When the function is called, it returns the 20 bytes numbered c_i to $\mathsf{c}_i + 19$ from the stream pointed to by `i`. It then sets $\mathsf{c}_i = \mathsf{c}_i + 20$, and advances `i` in a round-robin fashion.

2.3.2 *Initialization and Update.*

Initializing R *and* State. The WRNG does not explicitly initialize R and State. However, as with any other stack parameter that is not initialized by the program, these variables are implicitly initialized with the latest values stored in the memory address allocated to them. In Section 4, we provide some analysis

Table I. Entropy Sources

Source	Size in Bytes Requested
CircularHash	256
KSecDD	256
GetCurrentProcessID()	8
GetCurrentThreadID()	8
GetTickCount()	8
GetLocalTime()	16
QueryPerformanceCounter()	24
GlobalMemoryStatus()	16
GetDiskFreeSpace()	40
GetComputerName()	16
GetUserName()	257
GetCursorPos()	8
GetMessageTime()	16
NTQuerySystemInformation calls	
ProcessorTimes	48
Performance	312
Exception	16
Lookaside	32
ProcessorStatistics	up to the remaining length
ProcessesAndThreads	up to the remaining length

of the actual values with which these variables are initialized and note that they are highly correlated. We are not sure about the reason for this use of uninitialized variables.

Initializing and refreshing each instance of RC4. All instances of RC4 are initialized and refreshed by the same mechanism, which collects system entropy and uses it to rekey an RC4 instance. The collected system entropy is composed of up to 3,584 bytes of data from different operating system sources. Entropy collection is synchronous and is only done when an RC4 stream is initialized, or reaches the 16KB threshold. Table I lists the different operating system entropy sources. These sources include different OS query functions, including common functions such as `GetLocalTime` and `GetUserName`, and the more specialized functions such as `NtQuerySystemInformation`. In addition, 256 bytes are taken from a special internal state named `CircularHash`. (This variable is updated in a simple and predictable way every time `get_next_20_rc4_bytes()` is called. The exact details of this update are irrelevant for our exposition.) A system device driver called `KSecDD` is additionally involved in the initialization. Documentation of the different OS functions is available in Howard and LeBlanc [2002]. The symbolic names for the `NtQuerySystemInformation` classes were recovered using dynamic code analysis (using the tool StraceNT,[5] which finds names in system header files). Dynamic code analysis also shows that some functions do not return the number of bytes requested: For example, the `NtQuerySystemInformation` class `ProcessorStatistics` returns only 16 bytes, and class `ProcessesAndThreads` returns 0 bytes.

[5]`http://www.intellectualheaven.com/default.asp?BH=projects&H=strace.htm`

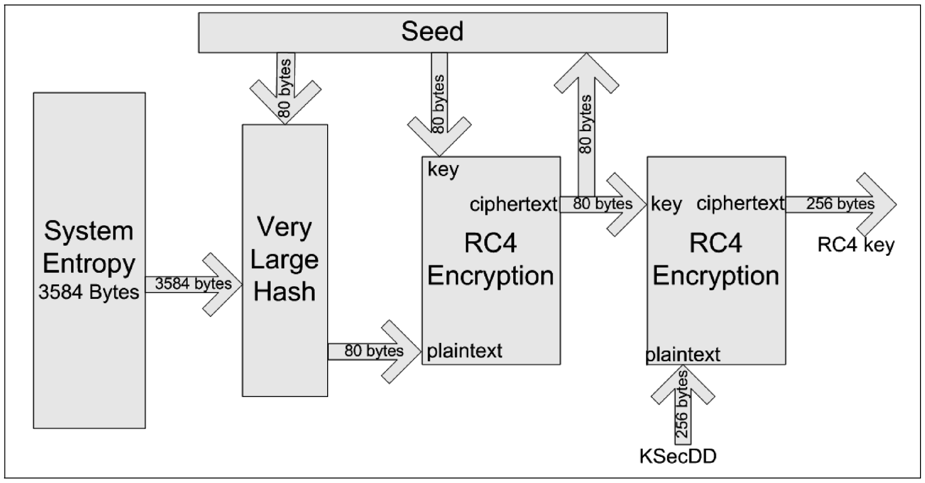

Fig. 3. RC4 rekeying function.

To summarize, the initialization and update of the RC4 states depend on many system variables. Some of these variables are predictable, but we were not able to find a practical brute force attack that predicts all 3,584 bytes of this input.

The pseudocode for the state refreshment mechanism is depicted in Figure 3. Every bit of the 3,584 bytes that were gathered is input to SHA-1 invocations, which affect the final result. In addition, this process uses a global 80-byte variable named seed, which is read from the registry, and data, which is read from a Windows device driver called KSecDD. In more detail, this initialization process is composed of the following stages.

— The entire 3,584 bytes of collected entropy, together with the 80-byte registry variable named seed, are hashed (using a function called VeryLargeHash) to produce an 80-byte digest. The function is implemented by a series of SHA-1 operations, designed to ensure that a change of a single input bit affects all output bits. The pseudocode of the function VeryLargeHash is presented in the following text. The Windows registry key seed is used by all instances of the WRNG run on the same machine and is stored at `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\RNG\Seed`, in the `HKEY_LOCAL_MACHINE` directory.
— The output of VeryLargeHash is fed into an RC4 encryption as a plaintext. The key to the encryption is the registry key seed.
— The result of the encryption is 80 bytes long. It is written to the registry variable seed and replaces its former value. It is also fed to another RC4 encryption as a key, and is used to encrypt an additional 256 bytes, which are read from a Windows device driver called KSecDD. This KSecDD device driver serves as an additional entropy source. The result is 256 bytes long.
— The result of the final encryption is used as a key for the RC4 instance that is used in the WRNG internal state. This RC4 instance is initialized using the RC4 key scheduling algorithm (KSA), described in Appendix A.

```
VeryLargeHash(Buf, Len, Seed) {
   k := Len / 4          // lengths are in bytes
   digest1 := SHA-1(Seed[00..19] | Buf[0..k − 1]
                        | Seed[20..39] | Buf[k..2k − 1])
   digest2 := SHA-1(Seed[20..39] | Buf[k..2k − 1]
                        | Seed[00..19] | Buf[0..k − 1])
   digest3 := SHA-1(Seed[40..59] | Buf[2k..3k − 1]
                        | Seed[60..79] | Buf[3k..4k − 1])
   digest4 := SHA-1(Seed[60..79] | Buf[3k..4k − 1]
                        | Seed[40..59] | Buf[2k..3k − 1])
   result[00..19] := SHA-1(digest1 | digest3)
   result[20..39] := SHA-1(digest2 | digest4)
   result[40..59] := SHA-1(digest3 | digest1)
   result[60..79] := SHA-1(digest4 | digest2)
   return result
}
```

Fig. 4. The function VeryLargeHash.

The Function VeryLargeHash. The algorithm of VeryLargeHash is described in Figure 4. It is based on a series of SHA-1 calls, performed on an input buffer (`Buf`) of any given length (`Len`), and a fixed-length 80-byte argument (`Seed`). (Note that this function uses standard SHA-1 rather than the SHA-1' variant, which is used in the main loop of the WRNG.) The buffer `Buf` holds the entropy buffer while `Seed` is the Windows global entropy parameter (which is common to all WRNG instances). The function defines the variable `k` to be equal to the length of the input buffer divided by 4, and operates on bytes `0` up to `4k-1` of `Buf`. Therefore, if the length of `Buf` is not divisible by 4, up to 3 bytes of `Buf` might be ignored. Any change to any other input byte affects all output bits.

Initializing all RC4 instances. The WRNG uses eight instances of RC4, all of which are initialized using the procedure described previously in the text. Initialization starts with the first call to read bytes from an instance. Note that the initializations of different RC4 instances used by one instance of the WRNG are run one right after the other; therefore, many of the 3,584 bytes of system parameters used for initialization will be equal in two successive initializations (still, we do not know how to use this fact to attack the initialization process).

Additional rekey calls of each of the eight RC4 instances are made after it outputs 16KB of data. (More accurately, an RC4 state whose accumulator exceeds 16KB threshold is not used to generate output, but is initialized and is skipped in the round-robin queue.) Therefore, all eight RC4 states are initialized simultaneously, one after the other, before any output is generated. The same applies to uninitialized states. Since there are eight RC4 instances, the generator always outputs $8 \times 16 = 128$KB of output between two rekey calls. (We ignore here some drift, which occurs because RC4 output is read in chunks of 20 bytes while $16{,}384 \mod 20 = 4$, which means that the refreshment

Table II. CryptGenRandom Component Scopes

Type	Location	Scope	Lifetime
Registry	System registry and optionally, hard drive	all users all processes	permanent
Kernel	Kernel memory space	all users all processes	from boot-up to shutdown
Static	Process memory space	single user single process	from first call to CryptGenRandom until the termination of the process
Volatile	Process stack	single user single thread	within one call to CryptGenRandom (from call to return)

of some states is shifted into the next 20-byte request from the `get_next_20_rc4_bytes`.)

2.4 Scope

Windows is running one WRNG instance per process. Therefore, two applications (e.g., Windows Word and Internet Explorer) have two separate states. The RC4 states and auxiliary variables of a specific process reside in DLL space, which is allocated on the first invocation of the Crypto API and remains allocated until it is unloaded. The state variables R and State, on the other hand, are stored on the stack. If a process has several threads, then they all share the same RC4 states stored in the DLL space, but each of them has its own stack and, therefore, its own copy of R and State.

It is interesting to note that R and State are never explicitly initialized, and instead are initialized with the last values that are stored in the stack locations allocated to them. In Section 4, we will describe an analysis that shows that there is correlation between the states used in different instances of the WRNG.

Scoping in more detail. Table II summarizes the different variable scopes in CryptGenRandom's code. These correspond to the standard scopes in the C programming language and in Windows. The variables used by the generator have different scope, as is detailed in the following text.

—*Registry variables*. CryptGenRandom uses one 80-byte long registry variable: `Seed`. CryptGenRandom uses it and updates its value when initializing the RC4 states, as is detailed in Section 2.3.2. It makes any changes to this variable permanent by setting a flag that causes it to be saved to the hard drive. This variable is accessible to all processes and users for reading and writing. Its exact location is `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\RNG\Seed`.

—*Kernel variables*. The `KSecDD` device driver is used in state initialization and refreshment. The output of this device is generated by a random-number generator closely related to the user-mode portion of CryptGenRandom itself. It is important to note that this device state is in kernel space and, therefore, is shared between all the users and processes on a machine.

```
; __stdcall FIPS186Gen(x,x,x,x,x,x,x)
_FIPS186Gen@28   proc near

SHA1context      = dword ptr -0B4h
SHA1context_buf  = dword ptr -9Ch
SHA1digest       = dword ptr -40h
State            = dword ptr -2Ch
R                = dword ptr -18h
local_pbuffer    = dword ptr -4
prov_field1      = dword ptr  0Ch
prov_field2      = dword ptr  10h
fips_IV          = dword ptr  14h
unused           = dword ptr  18h
pbuffer          = dword ptr  1Ch
dwlen            = dword ptr  20h

                 push    ebp
                 mov     ebp, esp
...
                 lea     eax, [ebp+R]
                 push    14h
                 push    eax
                 push    [ebp+prov_field2]
                 push    [ebp+prov_field1]
                 call    _NewGenRandom@16  ; NewGenRandom(x,x,x,x)
                 xor     eax, eax
```

Fig. 5. A snippet of the disassembly showing temporary variables (including `R` and `State`) on the stack.

—*Static variables*. These are allocated in fixed offsets in the data segment of the DLL. Their initial values are stored in the DLL's binary. Two important static variables are the array of eight RC4 states, and the `CircularHash` structure that is used in the initialization (see Section 2.3.2).

—*Volatile variables*. All other variables are local volatile variables, residing on stack. Figure 5 shows a snippet of the stack, including the variables `R` and `State`, from the disassembly that we performed. These variables reside quite deep in the stack—they are located 136 bytes from the beginning of the stack area allocated for the CryptGenRandom function call.[6] This fact will

[6]Address `0h` in Figure 5 is the EBP, that is, the stack base, of the function FIPS186Gen, which uses the variables `R` and `State`. The stack grows downward and `R` is in location `-18h` (i.e., 24 in decimal) in it. Now, the EBP of FIPS186Gen is at depth 112 bytes compared to the EBP of the function, which called CryptGenRandom. Thus, the full depth of `R` is 136 bytes. (The depth of the EBP of FIPS186Gen is not shown in Figure 5, since this figure shows a snippet of the static analysis of the code. A snippet of a debugger run would have shown this fact, but would be too long to be included in the article.)

The reader might notice that the function FIPS186Gen receives seven parameters, based on the stack layout and the actual parameter passing. However, this function ignores its first parameter, which, when invoked by CryptGenRandom, points to an unidentified field of the crypto provider object. We aim to keep the presentation short and, therefore, chose to omit this detail from Figure 5.

be relevant in the discussion in Section 4 on maintaining the state of these variables between calls.

Overall scope. All variables directly affecting the output of CryptGenRandom reside either on stack (`State, R`) or in the static area, namely the data segment of the DLL (RC4 states). Since the data segment of the DLL is loaded into the user-mode memory space, there is one set of static variables per process. There can be one set of stack variables per thread, assuming that different threads invoke CryptGenRandom. All variables that are shared on the machine (`Seed, KSecDD` state) participate only during the initialization and update of the RC4 states.

Scoping is both good and bad. On one hand, it separates between two processes, and, therefore, an attack on one WRNG (e.g., learning its state) does not affect applications using another WRNG. On the downside, the fact that there is only one consumer per WRNG, together with the very long period between rekeys, makes it very likely that the WRNG state will rarely be refreshed.

Implementation in user mode. The WRNG is running in user mode, rather than in the kernel. A kernel-based implementation would have kept the internal state of the WRNG hidden from applications, whereas a user mode implementation enables each process to access the state of the WRNG instance assigned to it.

3. ANALYSIS I: CRYPTANALYTIC ATTACKS

We argue here about the backward security of the generator and demonstrate an attack on its forward security. Namely, we show how an adversary that obtains the state of the WRNG (i.e., the values of the variables R and State and the states of the eight RC4 registers) is able to compute future and past states and outputs of the generator. Computing future states is easy. We show that it is also very easy to compute past states if the adversary knows the initial values of the variables State and R (but not of the RC4 instances). We also show two attacks that compute previous states without knowledge of the initial values of State and R. The computational overhead of these two attacks is 2^{40} and 2^{23}, respectively.

The attacks we describe can be applied by an adversary that learns the state of the generator. This is a very relevant threat for several reasons. First, new buffer overflow attacks are found each week. These attacks enable an adversary to capture the memory space of a certain process or of the entire computer system. Second, since the WRNG runs in user mode, a malicious user running an application can learn the WRNG state without violating any security privileges (this happens since the WRNG memory space is not blocked from that user).

3.1 Backward Security

Suppose that an adversary learns the state of the WRNG at a specific time. The next state of the WRNG, as well as its output, are a deterministic function of this data. The adversary can, therefore, easily compute the generator's output and its next state, using a simulation of the generator's algorithm (similar to the one we constructed). The adversary can then compute the following output and state of the simulator, as a function of the state it just computed. It can continue doing so until the next refresh of the generator using system entropy.

3.2 Attacks on Forward Security

The WRNG depends on RC4 for generating streams of pseudorandom output, which are then added to the state of the generator. RC4 is a good stream cipher, but it does not provide any forward security. Namely, given the current state of an RC4 cipher, it is easy to compute its previous states and previous outputs. (This process is described in Appendix A. See also Bellare and Yee [2003].) We use this fact to mount attacks on the forward security of the WRNG. Suppose an adversary learns the state of the generator at time t and wishes to compute the state at time $t-1$. We show here three methods of computing this state, with an overhead of a few steps, 2^{40} steps, and 2^{23} steps, respectively, and where the first attack also assumes knowledge of the initial values of State and R. The attack with an overhead of 2^{23} steps is based on observing that State is updated using consecutive addition and exclusive-or operations, which, up to the effect of carry bits, cancel each other.

An instant attack when the initial values of State and R are known. Suppose that the attacker knows the initial values of the variables State and R. (As argued in Section 4, this is a reasonable assumption.) The attacker also knows the current values of the eight RC4 registers. Since RC4 does not provide any forward security, the attacker can compute all previous states and outputs of the RC4 registers, until the first invocation of the WRNG. (It can learn the total number of invocations of the WRNG from a static variable named `stream_counter`, found in a static offset in memory—offset `7CA1FFA8` in the DLL of the version of Windows we examined.) Since each state of the WRNG is a function of the previous values of State and R and of the output of the RC4 registers, the attacker can now compute the states and outputs starting from the first step and continuing until the current time. We implemented this attack in the tool `PreviousCryptGenOutputs0`. The overhead of this attack is composed of computing RC4 and SHA-1' invocations. As the overhead of computing SHA-1' is greater, we will only consider it. Computing all 128KB of output requires about $2^{17}/20 \approx 2^{12.6}$ invocations of SHA-1'.

An attack with an overhead of 2^{40}. Let us denote by R^t and S^t the values of R and State just before the beginning of the tth iteration of the main loop. (We refer here to the main loop of the WRNG, as described in Figure 1.) Let us denote by $R^{t,i}$, $S^{t,i}$ the values just before the execution of the ith line of code in the tth iteration of the main loop (namely, $R^t = R^{t,4}$, $S^t = S^{t,4}$). Let RC^t denote the output of `get_next_20_rc4_bytes` in the tth iteration. Each of these values is 160 bits long. Let us also denote by X_L the left-most 120 bits of variable X (namely, its 120 most significant bits), and by X_R its 40 right-most bits.

Given R^t and S^t, our goal is to compute R^{t-1}, S^{t-1}. We also know the state of all eight RC4 registers, and since RC4 does not have any forward security, we can easily compute RC^{t-1}. We do not assume any knowledge of the output of the generator. We observe the following relations between the values of R and S before and after code lines in which they are changed:

$$\begin{aligned}
S^{t-1,11} &= S^t - R^t - 1 \\
R^{t-1,9} &= R^t_L \mid *^{40} \qquad \text{(where } *^{40} \text{ is a 40-bit string unknown at this stage)} \\
R^{t-1} &= R^{t-1,9} \oplus RC^{t-1} \\
S^{t-1} &= S^{t-1,5} = S^{t-1,11} \oplus R^{t-1,9} = \underbrace{(S^t - R^t - 1)}_{S^{t-1,11}} \oplus \underbrace{(R^t_L \mid R^{t-1,9}_R)}_{R^{t-1,9}}.
\end{aligned}$$

We also observe the following relation:

$$R^t_R = \text{SHA-1'}(S^{t-1,11})_R = \text{SHA-1'}(S^t - R^t - 1)_R.$$

These relations define R^{t-1}_L and S^{t-1}_L, but they do not reveal the rightmost 40 bits of these variables (namely R^{t-1}_R and S^{t-1}_R) and do not even enable us to verify whether a certain "guess" of these bits is correct. Let us, therefore, examine the previous iteration and, in particular, the process of generating R^{t-1}_R and use it to compute R^{t-1}_R (then, S^{t-1}_R can easily be computed).

$$\begin{aligned}
R^{t-1}_R &= \text{SHA-1'}(S^{t-2,11})_R \\
&= \text{SHA-1'}(S^{t-1} - R^{t-1} - 1)_R \\
&= \text{SHA-1'}(\underbrace{(S^t - R^t - 1) \oplus (R^t_L \mid R^{t-1,9}_R)}_{S^{t-1}} - \underbrace{(R^t_L \mid R^{t-1,9}_R) \oplus RC^{t-1}}_{R^{t-1}}) - 1)_R
\end{aligned}$$

Note also that $R^{t-1,9}_R = R^{t-1}_R \oplus RC^{t-1}_R$. Consequently, we know every value in this equation, except for R^{t-1}_R. We can, therefore, go over all 2^{40} possible values of R^{t-1}_R, and disregard any value for which this equality does not hold. For the correct value of R^{t-1}_R, the equality always holds, while for each of the remaining $2^{40} - 1$ values, it holds with probability 2^{-40} (assuming that the output of SHA-1 is uniformly distributed). We, therefore, expect to have a constant number of false positives, namely incorrect candidates for the value of R^{t-1}_R (see the following text for an analysis of the expected number of false positives after several invocations of this attack).

An attack with an overhead of 2^{23}. A close examination of the relation between the addition and exclusive-or operations reveals a more efficient attack. Note that $R^{t-1,9} = R^{t-1} \oplus RC^{t-1}$ and, therefore, as was shown earlier, the following relation holds:

$$R^{t-1}_R = \text{SHA-1'}(S^{t-2,11})_R = \text{SHA-1'}(S^{t-1} - R^{t-1} - 1)_R.$$

Note also that

$$S^{t-1} = \underbrace{(S^t - R^t - 1)}_{S^{t-1,11}} \oplus \underbrace{RC^{t-1} \oplus R^{t-1}}_{R^{t-1,9}}.$$

Let us use the notation $Z = (S^t - R^t - 1) \oplus RC^{t-1}$. We are interested in computing $R^{t-1}_R = \text{SHA-1'}((Z \oplus R^{t-1}) - R^{t-1} - 1)_R$. Denote by r_i the ith least significant bit of R^{t-1}. We know all of Z and the 120 leftmost bits of R^{t-1}, and should, therefore, enumerate over all possible values of the right-hand side of the equation, resulting from the 2^{40} possible values of $r_{39}, \ldots, r_0$. (We will see that typically there are much fewer than 2^{40} such values.)

Use the notation 0_Z and 1_Z to denote the locations of the bits of Z, which are equal to 0 and to 1, respectively.

$$\begin{aligned}(Z \oplus R^{t-1}) - R^{t-1} - 1 &= \left(\sum_{i\in 0_Z} 2^i r_i + \sum_{i\in 1_Z} 2^i(1-r_i)\right) - \sum_{i=0\ldots 159} 2^i r_i - 1\\ &= \sum_{i\in 1_Z} 2^i + \left(\sum_{i\in 0_Z} 2^i r_i - \sum_{i\in 1_Z} 2^i r_i - \sum_{i=0\ldots 159} 2^i r_i\right) - 1\\ &= Z - 2\cdot \sum_{i\in 1_Z} 2^i r_i - 1\\ &= Z - 2\cdot (R^{t-1} \wedge Z) - 1,\end{aligned}$$

where $\wedge$ denotes bit-wise AND. Therefore,

$$R_R^{t-1} = \text{SHA-1'}(Z - 2\cdot (R^{t-1} \wedge Z) - 1)_R.$$

The previous equation shows that the only bits of R^{t-1} that affect the result are bits r_i for which the corresponding bit z_i equals 1. The attack can, therefore, be more efficient. Consider, for example, the case that the 20 least significant bits of Z are 1, the next 20 bits are 0, and the other bits have arbitrary values. The attack enumerates over all 2^{20} options for $r_{19}, \ldots, r_0$. For each possible option, it computes the expression detailed earlier for R_R^{t-1}. It then compares the 20 least significant bits of the result to $r_{19}, \ldots, r_0$. If they are different, it disregards this value of $r_{19}, \ldots, r_0$, and if they are equal, it saves it. As before, the correct value is always retained, while each of the other $2^{20} - 1$ values is retained with probability $1/2^{20}$. We, therefore, expect a constant number of false positives.

In the general case, the attack enumerates over all possible values of the bits of R_R^{t-1}, which affect the result, namely r_i for which $0 \leq i \leq 39$ and $i \in 1_Z$. In case there are ℓ such bits, the attack takes 2^ℓ time. Therefore, assuming that Z is random, the expected overhead of the attack is $\sum_{\ell=0}^{40} 2^\ell \Pr(|1_{Z_R}| = \ell) = \sum_{\ell=0}^{40} 2^\ell \binom{40}{\ell} 2^{-40} = (3/2)^{40} \approx 2^{23}$. As before, the number of false positives is constant, since for every value of ℓ, we examine $2^\ell - 1$ incorrect values, and each one of them is retained with probability $2^{-\ell}$.

We implemented this attack in the tool `PreviousCryptGenOutputs23`. The average running time of recovering a previous state is about 19 seconds on a 2.80MHz Pentium IV (without any optimization). The tool can recover all previous states until the time the generator was initialized, as is detailed in the following text.

We note that there exist much faster implementations of SHA-1', and consequently of the attack. For example, recent experiments on the Sony PS3 machine show that on that platform it is possible to compute 86 million to 87 million invocations of SHA-1 per second (applying the function to 20-byte long inputs) [Osvik 2007]. In this implementation, computing 2^{23} invocations of SHA-1' should take less than 1/10 of a second. (The overall overhead of the attack is, of course, somewhat greater.)

Repeatedly applying the attack on forward security. The procedures detailed earlier provide a list of $O(1)$ candidate values for the state of the generator at time $t-1$. They can of course be applied again and again, revealing the states, and consequently the outputs, of the generator at times $t-1, t-2$, and so on. As for the number of false positives, whenever we apply the attack, 2^{23} attack, we have $2^{\ell}-1$ possible false positives, and each of them passes the test with probability $2^{-\ell}$ (the 2^{40} attack has $2^{40}-1$ potential false positives, and each of them passes the test with probability 2^{-40}). The analysis of this case is identical to the analysis of the number of false positives in an attack on the forward security of the Linux random number generator (see Gutterman et al. [2006], Appendix C). In that analysis, it was shown that the number of false positives can be modeled as a martingale and that its expected value at time $t-k$ is only k. (The number of false positives does not grow exponentially since for any false positive for the value of the state at time $t-k$, it happens with constant probability that the procedure detailed earlier does not result in any suggestion for the state at time $t-k-1$. In this case, we can dismiss this false positive and should not explore its preimages.)

Of course, if the attacker knows even a partial output of the generator at some previous time $t-k$, it can use this knowledge to identify the true state of the generator at that time and remove all false positives.

The effect of the attacks. The WRNG has no forward and backward security: An attacker that learns the state of the generator at time t can easily compute past and future states and outputs, until the times where the state is refreshed with system-based entropy. Computing all states and outputs from time t up to time $t+k$ can be done in $O(k)$ work (i.e., $O(k)$ invocations of SHA-1'). Computing candidates to all states and outputs from time t to time $t-k$ can be done by applying (an expected number of) about $2^{22}k^2$ SHA-1' invocations. (The $2^{22}k^2$ result is due to the fact that for every $1 \leq j \leq k$ we expect to find j candidate values for time $t-j$, and to each of these, we apply the 2^{23} attack to learn its predecessor. The total number of SHA-1' invocations is expected to be $\sum_{j=1}^{k} 2^{23} j \approx \frac{k^2}{2} 2^{23} = 2^{22}k^2$.) Computing L bytes of previous WRNG output requires going back $k = L/20$ steps. For $L = $ 1KB, 10KB, and 64KB, we get that SHA-1' needs to be invoked about 2^{33}, 2^{40}, and 2^{45} times, respectively. Using a single PS3 machine, these attacks should take about 100 seconds, 3.5 hours, and 5 days, respectively. The attack is fully parallelizable; therefore, usage of c machines reduces the attack time by a factor of c. Note also that our analysis of the usage of the WRNG shows that normal operation of Internet Explorer (and probably of other applications) does not consume a lot of WRNG output. We, therefore, do not expect that an attacker will need to compute a lot of previous outputs of the WRNG.

To summarize, an attacker that learns the state at time t can, therefore, apply this knowledge to learn all states of the generator in an "attack window", which lasts from the last refresh (or initialization) of the state before time t to the first refresh after time t.[7] As discussed previously in the text, the WRNG keeps a separate state per process, and this state is refreshed only after the generator generates 128KB of output. Therefore, we can make the following statement:

> Knowledge of the state of the generator at a single instance in time suffices to predict 128KB of its output. These random bits are used in the time period lasting from the last entropy refresh before the attack to the first refresh after it.

In case of a process with low random bit consumption, this window might cover days of usage. In the case of Internet Explorer, we note in Section 4 that it might run 600 to 1,200 SSL connections before refreshing the state of its WRNG. This observation essentially means that, for most users, leakage of the state of the WRNG used by Internet Explorer reveals all SSL keys used by the browser between the time the computer is turned on and the time it is turned off.

An observation about state updates. The update of the variable State in the main loop is based on exclusive-oring and adding R. More precisely, let S^{t} denote the value of State at the beginning of the tth iteration of the loop. Then $\mathsf{S}^{t+1} = (\mathsf{S}^{t} \oplus \mathsf{R}) + \mathsf{R}' + 1$, where R' is identical to R, except for the five least significant bytes, which are replaced with bytes from the output of the WRNG (which might be known to an attacker). The addition and exclusive-or operations are related (they are identical up to the effect of the carry, which affects addition but not the exclusive-or operation). Therefore, S^{t+1} is strongly related to S^{t} much more than if, say, it was defined as $\mathsf{S}^{t+1} = (\mathsf{S}^{t}, \mathsf{S}^{t} \oplus \mathsf{R}$. (Note, however, that we were not able to exploit these relations in order to attack the generator.)

4. ANALYSIS II: THE INTERACTION BETWEEN THE OPERATING SYSTEM AND THE GENERATOR

In the following text, we show how the generator is invoked by the operating system and how this affects its security.

Frequency of entropy-based rekeys of the state. Each process has its own copy of a WRNG instance. Since each instance of the WRNG uses eight RC4 streams, its state is refreshed, only after it generates 128KB of output. Between refreshes, the operation of the WRNG is deterministic. If one process (say, a web browser) uses very few pseudorandom bits, the WRNG instance that is used by this state will be refreshed very rarely, even if other processes consume many pseudorandom bits from the instances of the WRNG that they use. We described in Section 3 attacks on the forward and backward security of the WRNG that enable an attacker, which observes a state of the WRNG to learn all states and

[7]In general, forward security should be provided by the function that advances the generator, and the use of entropy to refresh the state of the generator is only intended to limit the effect of backward security attacks. In the case of the WRNG, the generator itself provides no forward security. Entropy-based refreshes, therefore, help in providing some limited forward security: The attack can only be applied until the last time the generator was refreshed.

outputs of the generator from the time it had its last refresh (or initialization) to the next time it will be refreshed.

Entropy-based rekeys in Internet Explorer. We examined the usage of the WRNG by Internet Explorer (IE), which might be the most security sensitive application run by most users (all experiments were applied to IE 6, version 6.0.2800.1106). The examination of Internet Explorer was conducted by hooking all calls to CryptGenRandom using a kernel debugger, and recording the caller and the number of bytes produced. When IE invokes SSL, it calls the WRNG through LSASS.EXE, the system security service, which is the only service used by IE for this purpose (as mentioned earlier, as a service LSASS.EXE keeps its own state of the WRNG). During an SSL session, there are a varying number of requests (typically, four or more requests) for random bytes. Each request asks for 8, 16, or 28 bytes at a time. We can, therefore, estimate that each SSL connection consumes about 100 to 200 bytes of output from the WRNG. This means that the instance of the WRNG used by IE asks for a refresh only after handling about 600 to 1,200 different SSL connections. It is hard to imagine that normal users run this number of SSL connections between the time they turn on their computer and the time they turn it off. Therefore, the attacks presented in Section 3 can essentially learn encryption keys used in all previous and future SSL connections of the attacked PC.

Initializing State and R. The variables State and R are not explicitly initialized by the generator, but rather take the last value stored in the stack location in which they are defined. This means that in many circumstances, these values can be guessed by an attacker knowledgeable in the Windows operating system. This is particularly true if the attacker studies a particular application, such as SSL or SSH, and learns the distribution of the initial values of these variables. Knowledge of these values enables an instant attack on the generator, which is even more efficient than the 2^{23} attack we describe (see Section 3).

We performed some experiments in which we examined the initial values of State and R when the generator is invoked by Internet Explorer. The results are as follows: (i) In the first experiment, IE was started after rebooting the system. In different invocations of the experiment, the variables State and R were mapped to different locations in the stack, but their initial values were correlated. (We did not examine the correlation in detail, but can say that the Hamming distances between the different initial values is small. For example, 13 of the 20 bytes of R, and 14 of the bytes of State, were equal in all invocations.) (ii) In the second experiment, IE was restarted 20 times without rebooting the system (no program was started between different invocations of IE). All invocations had the same initial values of State and R. (iii) In the third experiment, we ran 20 sessions of IE in parallel. The initial values of the variables were highly correlated (in all invocations but one, the initial value was within a Hamming distance of 10 bits or less from the initial value of another invocation). These results of our limited set of experiments might not suffice for an actual efficient brute-force attack on the initial values of State and R, but they do demonstrate that these values are far from random. More analysis might reveal stronger correlations and relations to other parameters and processes of the operating system, which might be used for an attack.

Maintaining the state of State *and* R. The variables State and R are maintained on the stack. If the WRNG is called several times by the same process, these variables are not kept in static memory between invocations of the WRNG, but are assigned to locations on the stack each time the WRNG is called (in this respect, they are different from the RC4 states, which are kept in static memory and retain their state between invocations). If State and R are mapped to the same stack locations in two successive WRNG invocations and these locations were not overwritten between invocations, then the variables retain their state. Otherwise, the variables in the new invocation obtain whatever value is on the stack. During the execution of the WRNG procedure (CryptGenRandom), these variables are updated according to the WRNG's algorithm. Recall that State and R are located quite deep in the stack (136 bytes deep, as is described in Section 2.4); therefore, we can hope that if the stack locations of State and R do not change between invocations, then no other function call changes the values in these locations. We performed several initial experiments to examine this issue. We used kernel debugger scripts, which examined the values of State and R whenever the WRNG is called by IE. In all but a few invocations, these variables were assigned to the same stack locations, and in these cases, they retained their values between invocations (namely, the value they had when exiting invocation j was equal to the value they had when entering invocation $j + 1$). A few times, the variables were assigned to different locations on the stack, and as a result, their values were not retained between successive invocations of the WRNG. (We noted that in all these cases, many bytes of the variables had the same initial value when entering the WRNG. This property might be caused by the fact that in these cases the WRNG is not called directly by IE, but rather by some other internal function, which was called by IE. This function changes the stack location of the WRNG's local variables but also has a side effect of writing these locations first with some constant values.)

We do not know how to explain this "loose" management of the state, and cannot tell whether it is a feature or a bug. In the attacks we describe in Section 3, we show how to compute previous states assuming that State and R retain their state between invocations of the generator. These attacks are relevant, even given the behavior we inspected earlier, for two reasons: (i) We observed that in IE the WRNG almost always retains the values of State and R. When it does not, the values of these variables seem to be rather predictable. The attacker can, therefore, continue with the attack until it notices that it cannot reproduce the WRNG output anymore. The attacker should then enumerate over the most likely values of State and R until it can continue the attack. (This attack requires an additional analysis of State and R, but it does seem feasible.) (ii) Other applications might use the WRNG in such a way that the stack locations in which the values of State and R are stored are never overwritten. (The following event might also happen: the variables State and R are assigned to stack locations that are different from their original locations, and are then assigned back to their original locations. This happens before the values in the original locations are corrupted. In this case, State and R obtain their previous values, and the attacker can attempt to resume the attack by skipping the output of the WRNG, which was generated using the new locations of State and R.)

Initialization of RC4 states. As noted previously, the different RC4 instances used by the same WRNG instance are initialized one after the other by vectors of system data, which are quite correlated. This is also true, to a lesser extent, for two instances of the WRNG run by two processes. On the other hand, the VeryLargeHash function that is applied to these values is based on the SHA-1' hash function and is likely to destroy any correlation between related inputs. We have not examined the entropy sources in detail, and we have not found any potential correlation of the outputs of the VeryLargeHash function.

The output of VeryLargeHash is used as a key for two RC4 encryptions of the variables Seed and KSecDD, respectively, and the result is used to initialize the RC4 state of the WRNG. Even if an attacker knows the values of Seed and KSecDD, they do not help it to predict the output of VeryLargeHash, and consequently, predict the initialization of the RC4 state. The RC4 algorithm itself is known to be vulnerable to related-key attacks, and it is known that its first output bytes are not uniformly distributed [Fluhrer et al. 2001]. We were not able, however, to use these observations to attack the WRNG, since it applies SHA-1' to its state before outputting it.[8]

Although we were not able to attack the RC4 initialization process, it seems that a more reasonable initialization procedure would have gathered system entropy once and used it to generate initialization data to all eight RC4 instances. (Say, by running the final invocation of RC4 in the initialization procedure to generate $8 \times 256 = 2{,}048$ bytes, which initialize all eight RC4 instances.)

Protecting the state. As the WRNG is running in user space (and not in protected kernel space), an adversary that wishes to learn the state of a certain application needs only break into the address space of the specific application. This property increases the risk of an attacker learning the state of the WRNG and consequently applying attacks on forward and backward security. (The WRNG is run in user space since each application is running its own WRNG copy. The other option would have been to let the system run a single generator in the kernel and use it to provide output to all applications.)

5. ANALYSIS OF THE UPDATE OF STATE

In the following section, we describe an analysis of the update of the variable State in the main loop of the generator (the main loop is described in Figure 1). We were not able to use the results of the analysis in order to attack the WRNG. Nevertheless, we report it here hoping that it might be useful for future research.

[8]We note that the distribution of the first output bytes of RC4 is known to be slightly biased, and the output of the WRNG is computed by applying SHA-1' to a function of RC4, State, and R. Therefore, an attacker that knows the values of State and R knows that there is a slight bias in the distribution of the output of the WRNG. However, this bias seems to be too weak to be useful.

The update of State is based on exclusive-oring and adding the variable R to State. To simplify the analysis, we first make the assumption that the five least significant bytes of R are not set to be equal to the output of the WRNG (i.e., line 7 in Figure 1 is removed). Denote the ith bit of State as s_i and the ith bit of R as r_i. Let 0_S define the set of indices for which $s_i = 0$. State is advanced in the following way:

$$\begin{aligned} \mathsf{State} &= (\mathsf{State} \oplus \mathsf{R}) + \mathsf{R} + 1 \\ &= \left(\sum_{i \in 0_S} 2^i r_i + \sum_{i \notin 0_S} 2^i (1 - r_i)\right) + \sum_{i=0\ldots159} 2^i r_i + 1 \\ &= \mathsf{State} + 2 \cdot \sum_{i \in 0_S} 2^i r_i + 1 \\ &= \mathsf{State} + 2 \cdot (\mathsf{R} \ \wedge \ \neg\mathsf{State}) + 1 \end{aligned}$$

(where $\wedge$ denotes bit-wise AND, and $\neg$ denotes bit-wise negation.)

Now, let us analyze the actual update of State. Denote by $\mathsf{State_L}$ the 120 most significant bits of Stateand by $\mathsf{State_R}$ its 40 least significant bits. Denote $\mathsf{R_L}$, $\mathsf{R_R}$ and out_R similarly. Then State is advanced in the following way (where '|' denotes concatenation):

$$\mathsf{State_L} \mid \mathsf{State_R} = [\mathsf{State_L} + 2 \cdot (\mathsf{R_L} \ \wedge \ \neg\mathsf{State_L})] \underbrace{\ \mid\ }_{\leftarrow \text{carry}} [(\mathsf{State_R} \oplus \mathsf{R_R}) + \mathsf{out_R} + 1]$$

Note that the only bits of R_L that affect the result are those that correspond to 0 bits of State. In addition, the carry bit resulting from the addition of rightmost 40 bits might affect the left 120 bits.

As for R, it is updated in the following way, where out is the output of the generator, and RC is the value returned by the function `get_next_20_rc4_bytes`, which advances the RC4 ciphers.

$$\mathsf{R_L}|\mathsf{R_R} \ = \ [\mathsf{R_L} \oplus \mathsf{RC_L}] \mid \mathsf{out_R}$$

Note that an observer which has access to the output of the generator knows the rightmost 40 bits of R.

The previous two observations demonstrate that some bits of R do not affect the update of State. On average, only half the bits of R (or R_L) affect the update of State. It is not clear, however, how to use these observations in order to attack the generator. Assume, for example, that we know $\mathsf{S^t}$, $\mathsf{R^t}$, the values of State and R at time t, that we also have access to the output of the generator, but that we do not know the output of the RC4 streams. Let $|0_{S_L}|$ denote the number of bits equal to 0 among the bits of $\mathsf{S_L^t}$. Then, in the next step, there are at most $2^{|0_{S_L}|+1}$ options for the value of $\mathsf{S_L^{t+1}}$ and 2^{40} possible values for $\mathsf{S_R^{t+1}}$. If $\mathsf{S^t}$ is random, we can expect only 2^{100} possible values of $\mathsf{S^{t+1}}$. We can make this number even lower if we can control the number of 0 bits in State to be low, as is possible if we can control the initialization of State (which is, as argued in Section 4, possible in many scenarios). As for R, there are 2^{120} options for $\mathsf{R_L^{t+1}}$, whereas knowledge of the output uniquely defines $\mathsf{R_R^{t+1}}$.

All this information does not help a lot. The output of the next iteration is defined as SHA-1'($S^{t+1} \oplus R^{t+1} \oplus RC^{t+1}$), and the unknown value of RC^{t+1} masks all the information we have about S^{t+1} and R^{t+1}. If we want to examine the number of possibilities for S^{t+2}, we have to apply the previous calculation starting from, say, 2^{60} possibilities for S_L^{t+1} and, therefore, end up with close to 2^{160} options for S^{t+2}.

6. COMPARISON WITH OTHER OPERATING SYSTEMS

6.1 Comparison with Windows XP

After the publication of a preliminary version of this work, Microsoft acknowledged that the random number generator of Windows XP, which is currently (by far) the most popular operating system, is vulnerable to our attacks on the forward security of Windows 2000.[9] However, Microsoft has not disclosed any details about the structure of the generator of XP. Microsoft has also stated that operating system versions starting with XP Service Pack 3 (SP3) and Vista SP1 use a different RNG design based on FIPS 800-90,[10] and are not vulnerable to our attacks.

We did a preliminary analysis of the code implementing CryptGenRandom in Windows XP Service Pack 2 (this was done before the announcement from Microsoft). It was possible to identify that the code implementing the external loop of the generator (the function `CryptGenRandom` of Figure 1) in XP is functionally identical to the code of Windows 2000. However, the code implementing the internal generation of pseudorandom numbers using RC4 (the function `get_next_rc4_bytes` of Figure 2) is different in XP. The size of this part of the code in XP is much greater than the size of the corresponding code in Windows 2000, and we have not analyzed the algorithm implemented by it. If this code implements a function, which, like RC4, does not provide forward security, then our attacks on forward security should also apply to Windows XP.

6.2 Comparison with the Linux PRNG

The pseudorandom number generator used in the Linux operating system (denoted LRNG) was analyzed in Gutterman et al. [2006]. The analysis of the WRNG shows that it differs from the LRNG in several major design issues.

—*Kernel versus User mode.* The LRNG is implemented entirely in kernel mode, while a large part of the WRNG is running in user mode.

Security implication: An application that runs in Windows and uses the WRNG can read the entire state of the WRNG, while the LRNG is hidden from Linux applications. This means that, compared to Linux, it is easier for an attacker to obtain a state of the WRNG.

[9]See Computerworld. 2007. Microsoft confirms that XP contains random number generator bug. `http://www.computerworld.com/action/article.do?command=printArticleBasic&articleId=9048438` .

[10]See `http://technet.microsoft.com/en-us/library/cc749132.aspx`.

—*Reseeding timeout.* The LRNG is feeding the state with system-based entropy in every iteration and whenever system events happen, while the WRNG is reseeding its state only after generating 128KB of output.

—*Synchronization*. The collection of entropy in the LRNG is asynchronous: Whenever there is an entropy event the data is accumulated in the state of the generator. In the WRNG, the entropy is collected only for a short period of time before the state is reseeded. In the long period between reseedings, there is no entropy collection.

—*Scoping.* The LRNG runs a single copy of the generator that is shared among all users running on the same machine. In Windows, on the other hand, a different instance of the generator is run for every process on the machine.

—*Efficiency of attacks*. The best forward security attack on the LRNG requires 2^{64} work. The attack on the forward security of the WRNG is, therefore, more efficient by a factor of about 2^{40} (it has an overhead of 2^{23} steps compared to 2^{64}).

Security implication. The impact of the previous four properties is that attacks on forward and backward security are more severe when applied to the WRNG. The attacks are more efficient by 12 orders of magnitude. They reveal the outputs of the generator between consecutive reseedings, and these reseedings are much more rare in the case of the WRNG. In some cases, reseeding the LRNG happens every few seconds, while the WRNG is reseeded every few days, if it is reseeded at all.

—*Blocking*. The LRNG implements an entropy estimation counter and provides an interface to the LRNG, which is blocked from generating output when there is not enough system entropy within the generator (this interface is `/dev/random`; the `/dev/urandom` interface to the LRNG, on the other hand, never blocks). This leads to situations where the generator halts until sufficient system entropy is collected. Hence, this also leads to easy denial of service attacks when one consumer of pseudorandom bits can empty the system entropy pools and block other users. The WRNG does not use entropy measurements and is, therefore, not blocking.

Security implication. Unlike the LRNG, the WRNG is not vulnerable to denial of service attacks.

7. CONCLUSIONS

7.1 Conclusions

WRNG design. The article presents a clear description of the WRNG, the most frequently used PRNG. The WRNG has a complex layered architecture, which includes entropy rekeying every 128KB of output, and uses RC4 and SHA-1' as building blocks. Windows runs the WRNG in user space and keeps a different instance of the generator for every process.

Attacks. The WRNG depends on the use of RC4, which does not provide any forward security. We used this fact to show how an adversary that learns the state of the WRNG can compute past outputs of the generator at an overhead of 2^{23} per output. The attacker can also learn future outputs by simulating the operation of the generator. These attacks can be run within seconds on a modern PC and enable such an attacker to learn the values of cryptographic keys generated by the generator. The attacks on both forward and backward security reveal all outputs until the time the generator is rekeyed with system entropy. Given the way in which the operating system runs the generator, this means that a single attack reveals 128KB of generator output for every process.

Code analysis. Our research is based on studying the WRNG by examining its binary code. We were not provided with any help from Microsoft and were only using the binary versions of Windows. To verify our findings, we developed a user mode simulator that captures WRNG states and computes future and past outputs of the WRNG. We validated the simulator output against real runs of the WRNG.

WRNG versus LRNG. We compared the pseudorandom generators used by Windows and Linux (WRNG vs. LRNG). The forward security attack on the WRNG is faster by a factor of 2^{40} compared to the attack on the LRNG. In addition, our findings show that the LRNG has better usage of operating system entropy, uses asynchronous entropy feedings, uses the extraction process as an entropy source, and shares its output between multiple processes. As a result, a forward security attack on the WRNG reveals longer sequences of generator output, compared to an attack on the LRNG.

7.2 Recommendations

Forward security. The most obvious recommendation is to change the algorithm used by the WRNG to one which provides forward security. This can be done by making local changes to the current implementation of the generator, or by replacing RC4 with a function that provides forward security. Alternatively, it is possible to use the transformation of Bellare and Yee [2003] which transforms any standard generator to one providing forward security. We believe, however, that it is preferable to replace the entire algorithm, used by the generator with a simpler algorithm, which is rigorously analyzed. A good approach is to adopt the Barak-Halevi construction. That construction, suggested by Barak and Halevi [2005], is a simple yet powerful construction of entropy-based PRNGs. Its design is much simpler to implement than the current WRNG implementation, and under the assumption that its building blocks are secure, it provably preserves both forward and backward security. It can be implemented using, say, AES and a simple entropy extractor.

Frequency of entropy-based rekeys. The generator should rekey its state more often. We also suggest that rekeys are forced based on the amount of time that has passed since the last rekey. It is important to note that entropy-based rekeys are required in order to limit the effect of attacks mounted by an adversary that obtains the state of the generator. (In a good generator, forward security and pseudorandomness are guaranteed by the function that advances the state and are ensured even if the generator generates megabytes or gigabytes of output between rekeys.) The risk of an adversary getting a hold of the state seems to be more dependent on the amount of time the system runs, than on the length of the output of the generator. Therefore, it makes sense to force rekeys every some time interval, rather than deciding whether to rekey based on the amount of output produced by the generator.

7.3 Open Problems

Extending our research to additional Windows platforms. Our entire research was conducted on a specific Windows 2000 build. We did several early checks on additional binary versions of Windows but that work is only in its beginning. The important operating systems to examine are the main Windows releases, such as Windows XP and Windows Vista, as well as systems that have fewer sources of entropy, such as Windows CE.

State initialization. As we stated in our analysis, the internal RC4 states are initialized and rekeyed with very similar entropy parameters. These are hashed by a procedure which uses SHA-1' and propagates a change in the value of a single input bit to all output bits. The result of this procedure initializes the RC4 algorithm. We were not able to use this finding, but it seems that additional research is needed here. The research should examine the different entropy sources and the hashing algorithm, and check if they result in any related-key attack on RC4. We also noted that the state variables State and R are not explicitly initialized but rather take the current values stored in the stack. More research is needed to examine in detail the distribution of these values.

APPENDIX

A. RC4

RC4 is a stream cipher. Its initialization process is defined in Figure A.1. The process of generating output is defined in Figure A.2.

RC4 has no forward security. Suppose we are given its state just before the tth iteration of the output generation algorithm (namely, the values of S^t, i^t and j^t). It is easy to compute the previous state, and consequently the previous output, by running the following operations:

```
for i from 0 to 255
    S[i] := i
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap(S[i],S[j])
```

Fig. A.1. RC4 key scheduling algorithm (KSA). The array *key* holds the key, *keylength* is the key size in bytes.

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap(S[i],S[j])
    output S[(S[i] + S[j]) mod 256]
```

Fig. A.2. RC4 pseudorandom generation algorithm. The output is xored with the clear text for encryption.

```
swap(S[i],S[j])
j := (j - S[i]) mod 256
i := (i - 1) mod 256
```

Therefore, given the state of RC4 at a specific time, it is easy to compute all its previous states and outputs.

ACKNOWLEDGMENTS

We would like to thank Dag Arne Osvik, Werner Schindler, and the anonymous referees for their helpful comments regarding this article.

REFERENCES

Barak, B. and Halevi, S. 2005. An architecture for robust pseudo-random generation and applications to /dev/random. In *Proceedings of the ACM Conference on Computing and Communication Security (CCS'05)*. ACM, New York.

Beaver, D. and Haber, S. 1992. Cryptographic protocols provably secure against dynamic adversaries. In *Advances in Cryptology (EUROCRYPT'92)*. Springer-Verlag, Berlin, 307–323.

Bellare, M. and Yee, B. S. 2003. Forward-Security in private-key cryptography. In *Proceedings of the Cryptographers' Track RSA Conference (CT-RSA)*. Springer, Berlin, 1–18.

Blum, L., Blum, M., and Shub, M. 1983. Comparison of two pseudo-random number generators. In *Advances in Cryptology (CRYPTO'82)*. Plenum Press, New York, 61–78.

Bundesamt für Sicherheit in der Informationstechnik. 1999. AIS 20: Functionality classes and evaluation methodology for deterministic random number generators. Tech. rep. https://www.bsi.bund.de/cae/servlet/contentblob/478130/publicationFile/30547/ais31e_pdf.pdf

Castejon-Amenedo, J. and McCue, R. 2003. Extracting randomness from external interrupts. In *Proceedings of the International Conference on Communication, Network, and Information Security*. International Association of Science and Technology for Development, Alberta, Canada, 141–146.

de Raadt, T., Hallqvist, N., Grabowski, A., Keromytis, A. D., and Provos, N. 1999. Cryptography in OpenBSD: An overview. In *Proceedings of the Annual USENIX Technical Conference.* USENIX, Berkeley, CA, 93–101.

Eilam, E. 2005. *Reversing: Secrets of Reverse Engineering*. Wiley, New York.

Ferguson, N. and Schneier, B. 2003. *Practical Cryptography*. John Wiley & Sons, New York.

Fluhrer, S. R., Mantin, I., and Shamir, A. 2001. Weaknesses in the key scheduling algorithm of RC4. In *Proceedings of the 8th Annual International Workshop on Selected Areas in Cryptography (SAC'01)*. Springer-Verlag, Berlin, 1–24.

Goldberg, I. and Wagner, D. 1996. Randomness in the Netscape browser. *Dr. Dobb's J*.

Guilfanov, I. 2006. The IDA Pro Disassembler and Debugger version 5.0. http://www.datarescue.com/idabase.

GUTMANN, P. 1998. Software generation of practically strong random numbers. In *Proceedings of 7th USENIX Security Symposium.* USENIX, Berkeley, CA.

GUTMANN, P. 2004. Testing issues with OS-based entropy sources. http://www.cs.auckland.ac.nz/~pgut001/pubs/nist_rng.pdf

GUTTERMAN, Z. AND MALKHI, D. 2005. Hold your sessions: An attack on Java session-Id generation. In *Proceedings of the Cryptographers' Track RSA Conference (CT-RSA)*. Springer, Berlin, 44–57.

GUTTERMAN, Z., PINKAS, B., AND REINMAN, T. 2006. Analysis of the Linux random number generator. In *Proceedings of the IEEE Symposium on Security and Privacy Conference.* IEEE, Los Alamitos, CA, 371–385.

HOWARD, M. AND LEBLANC, D. 2002. *Writing Secure Code*, 2nd Ed. Microsoft Press, Redmond, WA.

KELSEY, J. 2004. Entropy and entropy sources in X9.82. http://csrc.nist.gov/CryptoToolkit/RNG/Workshop/EntropySources.pdf.

KELSEY, J., SCHNEIER, B., AND FERGUSON, N. 1999. Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography*. Springer, Berlin, 13–33.

KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. 1998. Cryptanalytic Attacks on Pseudorandom Number Generators. In *Proceedings of the 5th International Workshop on Fast Software Encryption*. Springer, Berlin, 168–188.

MICROSOFT. 2006. Debugging tools for Windows. http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

MURRAY, M. R. V. 2002. An implementation of the Yarrow PRNG for FreeBSD. In *Proceedings of BSDCon*, S. J. Leffler, Ed. USENIX, Berkeley, CA, 47–53.

OSVIK, D. A. 2007. Personal communication.

SHAMIR, A. 1981. On the generation of cryptographically strong pseudo-random sequences. In *Proceedings of the International Colloquium on Automata, Languages and Programming*. Springer, Berlin, 544–550.

TS'O, T. 1994. random.c | linux kernel random number generator. http://www.kernel.org.

YUSCHUK, O. 2004. OllyDbg 1.1: A 32-bit assembler level analysing debugger for Microsoft Windows. http://www.ollydbg.de/.

Received November 2007; revised August 2008, March 2009; accepted June 2009