The Sapphire II Stream Cipher

Encryption, Pseudorandom Number Generation, and Cryptographic Hash Generation

by Michael Paul Johnson

The Sapphire II Stream Cipher is a reasonably fast, compact, portable algorithm that is useful for encryption, authentication, and pseudorandom number generation. This algorithm fills a niche that makes it uniquely qualified in a variety of respects to fill some common needs for security. The balancing act between speed, size, portability, security, versatility, and exportability make the Sapphire II Stream Cipher a good choice for many applications. Because it is a relatively new stream cipher and a very new method of cryptographic hash generation, caution and further study is recommended before using it in critical security applications.

Background

There are many good encryption algorithms available that can be used to add security to software applications. There are also a lot of really poor excuses for cryptography in use, even by large, otherwise reputable software publishers. Naturally, there are some things that are in between. It takes some serious study to create a serious cryptosystem, and there are many issues besides the core algorithm to consider, such as key management, file formats, and use of temporary files and virtual RAM. The Sapphire II Stream Cipher should be a useful building block in an application that includes some security features.

The basic core of this stream cipher is the same as one I started work on in November 1993, which involves the constant shuffling of a permutation vector. I was surprised and somewhat amused to see a similar cipher (at least in the core idea of a permutation vector that was partially shuffled with each byte emitted) posted to the Internet news group sci.crypt as being compatible with RC4. This posting is available on the Internet at ftp://ftp.dsi.unimi.it/pub/security/crypt/code/rc4.revealed.gz and in the sci.crypt archives on ripem.msu.edu. The Sapphire II Stream Cipher is fundamentally different, however, in that it uses both plain text and cipher text feedback, and thus is both easier to use in a secure way. The Sapphire II Stream Cipher is much less vulnerable to a known plain text attack. See “Stream Cipher Structure,” below. The Sapphire II Stream Cipher differs from the original Sapphire Stream Cipher in that it has been made more robust against an adaptive chosen plain text attack.

Generic Stream Cipher Structure

There are a wide variety of stream ciphers, but almost all of them fit into a subset of the structure shown in figure 1.

The “public information” box represents things like the current time or a usenet news group that is available to both the sender and receiver. This input may make sense for live communications channels, but is not practical for file storage. The Sapphire II Stream Cipher doesn’t make use of this possible path.

The feedback function determines how the state variables are updated with each cycle of the stream generator. It may or may not make use of feedback from the plain text, cipher text or results of the output function. Making use of cipher text feedback improves data masking properties of the cipher. The plain text feedback improves the authentication properties of the cipher (making cryptographic hash generation possible), but reduces the cipher’s ability to recover meaningful data if the input stream is in error.

The output function determines what the next output unit should be based on the values of the state variables. An output unit may be a bit, a byte, or a machine word, but is normally not any longer than that.

The combination function is usually something simple, like addition modulo-2 (exclusive-or) or modulo-256 (byte-wise addition without carry).

Decryption is the same as encryption, except that the inverse of the combination function is used and the roles of plain text and cipher text feedback are reversed.

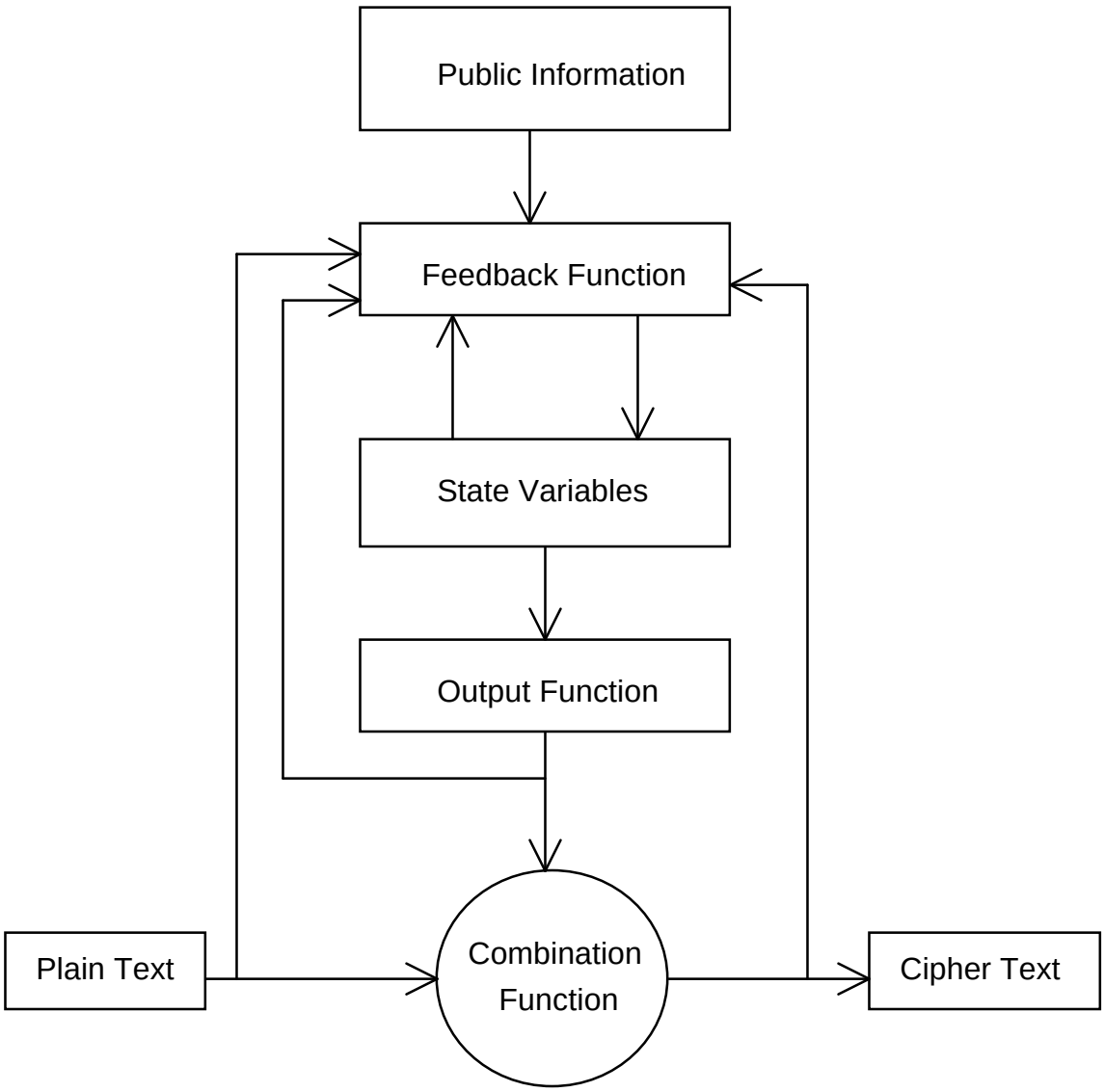

Figure 1. Generic stream cipher encryption. The stream cipher derives its cryptographic strength from the complexity of the output function or the complexity of the feedback function, or both. Not all data paths shown are used by all stream ciphers.

Overview of the Sapphire II Stream Cipher

The Sapphire II Stream Cipher is designed to have the following properties:

- Useful for generation of cryptographic check values as well as protecting message privacy.
- Accepts a variable length key for strength control.
- Strong enough to justify *at least* a 64 bit key for balanced security. Further study is recommended to see if this is, in fact, the case.
- Small enough to be built into other applications with several keys active at once.

- Key setup fast enough to support frequent key change operations but slow enough to discourage brute force attack on the key.
- Fast enough to not significantly impact the speed of file read & write operations on most current platforms.
- Portable among common computers and efficient in C, C++, and Pascal.
- Operates on one 8-bit byte at a time.
- Include both cipher text and plain text feedback (for both optimal data hiding and value in creation of cryptographic check values).
- Acceptable performance as a pure pseudorandom number generator.
- Allows some key re-use without severe security impacts.

The Sapphire II Stream Cipher is based on a state machine. The state consists of 5 index values and a permutation vector. The permutation vector is simply an array containing a permutation of the numbers from 0 through 255. Up to four of the bytes in the permutation vector are moved to new locations (which may be the same as the old location) for every byte output. The output byte is a nonlinear function of all 5 of the index values 8 of the bytes in the permutation vector, thus frustrating attempts to solve for the state variables based on past output. On initialization, the index variables are set (somewhat arbitrarily) to the contents of the permutation vector at locations 1, 3, 5, 7, and a key-dependent value. The permutation vector (called the cards array in the source code) is shuffled based on the user's key. This shuffling is done in a way that is designed to minimize the bias in the destinations of the bytes in the array. The biggest advantage in this method is not in the elimination of the bias, per se, but in slowing down the process slightly to make brute force attack more expensive.

Key Setup

Key setup (illustrated by the function initialize in sapphire.cpp and sapphire.pas) consists of three parts:

1. Set the permutation vector to a known state (a simple counting sequence).

2. Starting at the end of the vector, swap each element of the permutation vector with an element indexed somewhere from 0 to the current index (chosen by the function keyrand).

3. Initialize the index variables to key-dependent values.

The keyrand function (see sapphire.cpp or sapphire.pas) returns a value between 0 and some maximum number based on the user's key, the current state of the permutation vector, and an index running sum called rsum. Note that the length of the key is used in keyrand, too, so that a key like "abcd" will not result in the same permutation as a key like "abcdabcd".

If this instance of the stream cipher is going to be used for cryptographic check value (hash) generation and not for encryption, a separate, faster initialization is used that simply initializes the indices as above and sets the permutation vector to an inverse counting sequence. See the function hash_init in sapphire.cpp or sapphire.pas.

Encryption

Each encryption operation involves updating the index values, moving (up to) 5 bytes around in the permutation vector, selecting an output byte, and adding the output byte bitwise modulo-2 (exclusive-or) to the plain text byte to produce the cipher text byte. The index values are incremented by different rules. The index called rotor just increases by one (modulo 256) each time. Ratchet increases by the value in the permutation vector pointed to by rotor. Avalanche increases by the value in the permutation vector pointed to by another byte in the permutation vector pointed to by the last cipher text byte. The last plain text and the last cipher text bytes are also kept as index variables. See the function encrypt in sapphire.cpp or sapphire.pas.

Decryption

Decryption is the almost same as encryption, except for the assignments to last_plain and last_cipher, which are swapped. The return value is the plain text byte instead of the cipher text byte. See the function decrypt, in sapphire.cpp or sapphire.pas.

Pseudorandom Number Generation

If you want to generate random numbers without encrypting any particular cipher text, simply encrypt 0 (or any other arbitrary value). There is still plenty of complexity left in the system to ensure unpredictability (if the key is not known) of the output stream when this simplification is made. If you want even less predictable numbers, you could encrypt some physical information that may be available to you, like keystroke timings or the number of processes running on a multi-user system. The quality of the pseudorandom numbers obtained in this way are much better for statistical modeling and even games than the linear congruential generators usually found as a library function.

Hash Generation

A cryptographic check value (also called a hash or message integrity check value) is simply a “finger print” of a message computed in such a way that it is computationally infeasible to find another message that has the same “finger print.” Such hash values are commonly used with digital signatures, since all currently known digital signature algorithms are much to slow to apply to an entire message. They are also useful for checking that a file has not been modified by a virus or other malicious agent. A cryptographic hash takes longer to generate than a CRC, but it

is trivial to modify a file to have any desired CRC. To use the Sapphire II Stream Cipher to compute a cryptographic check value of a message of arbitrary length:

1. Initialize the stream cipher state with a initialize or hash_init.

2. Encrypt all of the bytes of the message to check (or if the message was encrypted, decrypt all of the bytes).

3. Call hash_final to get the hash value.

The hash_final function “stirs” up the permutation vector by encrypting an inverse counting sequence from 255 down to 0 to ensure that the effects of the last few bytes of the message are well distributed. It then encrypts zeroes to obtain the hash value. This step is essential in ensuring that the effect of the last byte has fully “avalanched” its effect throughout the output bytes. The normal length of a cryptographic hash is 20 bytes, but it would be reasonable to use any value from 16 to 32. 16-byte hash values are generated by some older, well-established cryptographic hash functions like MD5. 20-byte hashes (like those generated by NIST’s SHA) are long enough to be immune to brute force attacks in the foreseeable future, and a good balance between space requirements and security. A 32-byte hash is suitable for very long term security, provided that nobody finds an attack better than brute force on this hash generation method.

The primary advantage of using the Sapphire II Stream Cipher for hash generation instead of established standards like SHA or MD5 is speed. This is especially true if you were going to use the Sapphire II Stream Cipher for encryption, too. The disadvantage relative to the more established options is that the worthiness of the Sapphire II Stream Cipher as a cryptographic hash function has not been evaluated nearly as heavily as SHA or MD5.

Key Size Selection

There are really two kinds of user keys to consider: (1) random binary keys, and (2) pass phrases. Analysis of random binary keys is fairly straight forward. Pass phrases tend to have much less entropy per byte, but the analysis made for random binary keys applies to the entropy in the pass phrase. The length limit of the key (255 bytes) is adequate to allow a pass phrase with enough entropy to be considered strong.

To be real generous to a cryptanalyst, assume dedicated Sapphire II Stream Cipher cracking hardware. The constant portion of the key scheduling can be done in one cycle. That leaves at least 256 cycles to do the swapping (probably more, because of the intricacies of keyrand, but we'll ignore that, too, for now). Assume a machine clock of about 256 megahertz (fairly generous). That comes to about one key tried per microsecond. On average, you only have to try half of the keys. Also assume that trying the key to see if it works can be pipelined, so that it doesn't add time to the estimate. Based on these assumptions (reasonable for major governments), and rounding to two significant digits, the following key length versus cracking time estimates result:

Key length, bits	Time to crack
32	35 minutes (exportable in Quicrypt)
33	1.2 hours (not exportable in Quicrypt)
40	6.4 days (exportable in RC4)
56	1,100 years (DES key size)
64	290,000 years (good enough for most things)
80	19 billion years (Skipjack's key size)
128	5.4×10^{24} years (Sufficient for very long term security)

Naturally, the above estimates can vary by several orders of magnitude based on what you assume for attacker's hardware, budget, and motivation. It would be possible to employ 10,000 cracking machines in parallel, for example, making 56-bit keys vulnerable to rapid cracking. Note that the larger key sizes (at least 64 bits) have lots of room for error in this estimation while still protecting data for its useful lifetime.

In the range listed above, the probability of spare keys (two keys resulting in the same initial permutation vector) is small enough to ignore, since the number of keys possible is much less than the number of possible states of the permutation vector ($2^{128} << 256!$).

Strength

There are several measures of strength for a stream cipher. There are other measures of strength for a cryptographic hash function. The Sapphire II Stream Cipher tries to measure as both a stream cipher and a cryptographic hash function, while still maintaining high speed and small size in a software implementation.

For a stream cipher, internal state space should be at least as big as the number of possible keys to be considered strong. The state associated with the permutation vector alone constitutes overkill, since 256! is much greater than a typical key space of 2^{64}.

If you have a history of stream output from initialization (or equivalently, previous known plaintext and ciphertext), then rotor, last_plain, and last_cipher are known to an attacker. The other two index values, flipper and avalanche, cannot be solved for without knowing the contents of parts of the permutation vector that change with each byte encrypted. Solving for the contents of the permutation vector by keeping track of the possible positions of the index variables and possible contents of the permutation vector at each byte position is not possible, since more variables than known values are generated at each iteration. Indeed, fewer index variables and swaps could be used to achieve security, here, if it were not for the hash requirements.

A more severe test of a stream cipher with feedback is to allow an attacker to choose inputs, see the outputs, and reorigninate the cipher at will. The attacker then tries to determine the internal state of the cipher or the corresponding key. The multiple index operations of the Sapphire II Stream Cipher's output function and the key-dependent initialization of the index variables are designed to make attacks that make use of this situation to determine the key or

predict future output impractical. It is possible, however, for an attacker to learn some things about the stream cipher's internal structure from such an attack. Then again, it would be easier to learn about the structure of the cipher by reading this article.

Another measure of strength is how long a cipher has with stood public scrutiny without a serious weakness being published. The wait for public scrutiny doesn't actually make the cipher stronger or weaker, it just increases confidence. On the other hand, the longer a cipher is known, the more likely that someone will find a weakness and *not* publish it. This is one measure that we will have to wait to see how the Sapphire II Stream Cipher fares, since it has only recently been released to the public.

One measure of the strength of a cryptographic hash function is that any small change in the plain text results in a change of approximately half of the bits of the resulting hash function. Although there exists a class of small two or three byte changes that result in only small changes the cipher text emitted, this situation is avoided by the post-processing done in the hash_final function (see sapphire.cpp or sapphire.pas).

Integrating Sapphire II with Your Applications

A trivial program to demonstrate both the encryption and hash generation capabilities of the Sapphire II Stream Cipher in Pascal is shown in stest.pas. A functionally identical program in C++ is shown in sapptest.cpp.

Exporting the Sapphire II Stream Cipher

The U. S. Department of State, Office of Defense Trade Controls (DOS/DTC), together with the National Security Administration, limit the strength of computer software that can be shipped out of the USA without a license from the DOS/DTC. The most frustrating thing about this is that they will not tell you what the limit is. Naturally, this raises some serious Constitutional questions. They will, however, review complete products that are ready for market in all respects and tell you if a license is required from the Department of State to export the product, or if the product can be exported under Department of Commerce rules (which are much more relaxed). They claim that they will complete such a review, under certain circumstances, in a maximum of 15 business days. they missed this deadline both times I tried the process, but I did learn some things that may be useful for you.

I wrote a shareware encryption program called Quicrypt that uses the Sapphire Stream Cipher (which is very close to the Sapphire II Stream Cipher). Quicrypt encrypts a random session key (from keystroke timings, among other sources of random data) with the user's key (a memorized pass phrase), then encrypts the user's data with the random session key. The random session key is limited to 32 bits in the exportable version, and 128 bits in the non-exportable version. The 32-bit session key limit may or may not depend on other factors in the Quicrypt design (like the fact that only the session key and not the user's key is limited in length), but you should have no trouble freely exporting any encryption product that uses such a short

cryptographic key if there is any fairness to the DOS/DTC and NSA at all. You might even be able to talk them into the use of a longer key, since 40-bit keys are routinely approved for RC4, which is similar in some ways. I later upgraded the domestic (non-exportable) version of Quicrypt to use Sapphire II instead of Sapphire, but left the exportable version as it was, since the improvement from Sapphire to Sapphire II makes essentially no difference with a weakened key.

There is a another way to weaken a cryptographic product for export without making it quite so vulnerable to criminal cryptanalysis. You could simply encrypt all or part of the session key with the public key of an escrow agent, and embed the result in the output file. I didn't do this, since it would involve royalties to Public Key Partners (not so bad if you were using RSA for something else, too, but I wasn't) and because selection of the escrow agent(s) involves some of the same sticky issues associated with the U. S. Government's Escrowed Encryption Standard. I didn't ask the NSA about this option specifically, but they seemed open to options similar to this.

Availability of Source Code

Source code is available on the internet as ftp://ftp.csn.net/mpj/I_will_not_export/crypto_???????/file/sapphire.zip, where the ??????? is revealed along with the export warning in ftp://ftp.csn.net/mpj/README. For more information on Quicrypt, mentioned above, get ftp://ftp.csn.net/mpj/qcrypt11.zip. Sapphire.zip and qcrypt11.zip are also available from the Colorado Catacombs BBS at 303-772-1062.

Challenge

Since the Sapphire II Stream Cipher is a new cipher, and the approach mentioned above for fast cryptographic hash generation is also new, you are invited to help evaluate the security of these two uses of this algorithm. If you find any weaknesses, please let me know at m.p.johnson@ieee.org or at PO Box 1151, Longmont CO 80502-1151, USA. I'll send a genuine sapphire to the first one to reveal to me a method better than brute force for solving for the key (or the state variables) of the Sapphire II Stream Cipher from known plain and cipher text. I'll also send a genuine sapphire to the first person to reveal to me a way better than brute force to manufacture a forged message with a given Sapphire II hash.

About the Author

Michael Johnson has been interested in computers and cryptography since he was a little boy. He earned an MSEE degree at the University of Colorado at Colorado Springs, where he wrote a thesis on data compression and cryptography.