Joint Hardware / Software Design of a Fast Stream Cipher

Craig S.K. Clapp

PictureTel Corporation, 100 Minuteman Rd., Andover, MA 01810, USA
craigc@pictel.com

Abstract. We explore the problem of designing a stream cipher that is fast in software yet may be efficiently implemented in hardware. We show that a keystream generator built as a word-wide non-linear-feedback shift register can offer both a high degree of parallelism and the hardware simplicity and flexible security of an iterated design. WAKE-ROFB is shown to be an example of this topology. A modified non-linear mixing function is proposed for WAKE-ROFB which makes it better suited to hardware implementation. The high degree of parallelism allows efficient implementation on processors having instruction-level parallelism, and leads naturally to high-speed pipelined hardware implementations. The recommended variant runs at 340 Mbps on a 266 MHz Pentium II and 270 Mbps on a 100 MHz TriMedia VLIW CPU, while a 2000 gate hardware implementation of the same cipher achieves 200 Mbps from a 50 MHz clock. A higher speed variant achieves 600 Mbps, 340 Mbps and 400 Mbps respectively with some loss of security, while needing slightly less hardware.

1 Introduction

The first Fast Software Encryption Workshop in December 1993 brought, perhaps for the first time, a major focus to the topic of cipher design for software based systems. The new degrees freedom afforded by this domain have led to many avenues of research that would previously not have been considered. Since then numerous ciphers have been proposed for software implementation, but of those that have the highest throughput in software none is especially well suited to hardware implementation. Typically they use large tables, e.g. SEAL [7], Blowfish [9], WAKE [10] and its derivatives [3], or have a large internal state, e.g. RC4 [8], STEPRIGHTUP [4].

In some applications there is a need for a high speed cipher that can be efficiently implemented in both hardware and software. One example comes from the trend to equip consumer electronics entertainment devices with high-speed digital interconnections.

High-speed digital interfaces, notably IEEE-1394, are expected to appear soon on digital video disc (DVD) players and digital television sets, and they have already appeared in personal computers. In response to the entertainment industry's concern to protect their content from unauthorized copying while

being transmitted digitally between consumer electronics devices, it has been proposed to encrypt such connections whenever they carry copyrighted data.

In consumer electronics devices whose workload does not justify a high performance processor, such as a DVD player that simply passes the compressed data from the disc to the external digital interface encrypting it along the way, it is preferred to implement the encryption in hardware.

When compressed digital video content is re-played on a personal computer the computationally intensive task of video decompression already dictates the presence of a high-performance CPU, typically one with architectural optimizations for media-processing. Given its presence it is natural to ask it to also assume the task of decryption. Digital televisions will increasingly use embedded media-processors for these same tasks, so efficiency on these is also a consideration.

The performance requirements of the aforementioned application motivate our interpretation of ‘high speed’ and ‘efficient hardware’. The consumer electronics industry’s goals for this application equate to a throughput in software in excess of two bits per cycle on a general purpose computer, and a hardware implementation in 1000 to 2000 gates.

We know of no reported ciphers that simultaneously achieve these goals.

2 Joint optimization for hardware and software

We anticipate that the twin goals of fast software implementation and economical hardware are not mutually exclusive as might be concluded from the lack of ciphers sharing both characteristics. Rather we suspect the reason for the lack of good examples is that there has been a polarization of design philosophies to favor one or other of the environments in the extreme. Our strategy is to design a cipher while jointly optimizing for the respective constraints arising from both hardware and software implementations.

By presenting this work we hope to stimulate further work in jointly optimizing ciphers for both hardware and software.

2.1 Trade-off between efficiency in hardware versus software

Our goal is to find a topology that runs at high speed in software while being very efficient in hardware. Moreover, we would ideally like it to run efficiently not only on legacy architectures such as the ‘x86, but also to take full advantage of the substantial instruction-level parallelism present in more advanced processors such as Intel’s Pentium II and media-processors such as Philips’ TriMedia chip.

The desire for a low gate-count tends to dictate a cipher with an efficient iterative implementation and relatively little state. Block ciphers such as DES typify such a strategy, but their iterated design tends to offer little parallelism that might be exploited in a software implementation.

Since even today’s multimedia-optimized processors are capable of performing four or five 32-bit operations per cycle (or 128 to 160 bit-operations per

cycle usable in a variety of word-lengths), it is highly desirable for a multimedia-optimized cipher to be capable of exploiting at least this much instruction-level parallelism. This tends to dictate that such a cipher have at least 128 to 160 bits of state, which of itself would consume a substantial proportion of our 1000 to 2000 gate hardware budget.

2.2 Fast software ciphers from hardware oriented ciphers

Many hardware oriented stream ciphers in the literature are shift-register based. However, characteristically these run very inefficiently in software since processors operate on words while these hardware oriented ciphers rely heavily on individual bit operations.

One way of utilizing the power of a word-oriented processor is to treat its words as n -bit vectors, and implement n independent copies of the cipher, one in each bit-lane, SIMD fashion. A variation on this theme is to allow some mild interaction between the bit lanes such as by replacing bit-wise addition of words (i.e. exclusive-OR) by regular addition of words, thereby converting n independent shift registers into an additive, or lagged Fibonacci, generator [5]. Such word-wide generators can then be used in combinations modeled after the favored topologies for combining bit-serial shift registers. Examples of this strategy can be seen in Fish [2], which is a cross between an additive generator and n concurrent shrinking generators [6], and Pike [1] which is a trio of additive generators whose bit-wise combination topology is patterned after A5 [8].

Common to these examples is that the increase in software speed comes at the expense of a corresponding increase in cipher state. Consequently these derivatives are no longer efficient in hardware. For instance, a 32-bit implementation of Fish has 3424 bits of state while Pike has 5440, a far cry from the few tens of bits of their hardware oriented prototypes.

An interesting question is whether by providing much stronger and non-linear interaction between the bits of the word (as opposed to the mild interaction in an additive generator) it is possible to achieve useful security with dramatically fewer register stages. In such a case, we can no longer expect the security to be established from some underlying single-bit shift-register cipher model.

2.3 Economical hardware ciphers from fast software ciphers

Taking fast software ciphers as our starting point, a natural inclination is to find a way to reduce their hardware complexity. Since for many such designs the hardware is dominated by look-up-tables an obvious question is how such look-up-tables might be reduced in size without crippling the cipher. An approach applicable to iterated block ciphers might be to increase the number of rounds to compensate for any loss of strength arising from reduced tables. However, there are no block ciphers at the speed we seek to achieve, leaving us to consider which of the fastest available stream ciphers might be modified to fit our needs.

Fast stream ciphers do not typically have a structure that supports incremental trade-off of speed versus security. A counter example comes from the

derivatives of WAKE described in [3]. These ciphers have as one parameter the total number of mixing/register stages. Increasing the number of stages increases the security. Unfortunately it also increases the total amount of state associated with the cipher, resulting in an additional hardware burden that we would rather not incur.

3 WAKE, WAKE-OFB, and WAKE-ROFB

In [10] Wheeler introduced the cipher WAKE. It uses a mixing function $\mathbf{M}$ that combines two 32-bit inputs into one 32-bit output with the aid of a key-dependent 256×32 -bit look-up-table, T . The particular construction of T makes the mixing function invertible in the sense that knowledge of the output word and one of the two input words is sufficient to uniquely specify the other input word.

WAKE consist of cascading four of these mixing functions having registered feedback around each one and overall feedback around the group. Four stages are chosen as the minimum number needed for complete diffusion but more stages can be added if desired for additional security.

As originally proposed, WAKE operated with cipher-feedback into the state machine, leading to the potential for chosen plaintext attacks as noted by Wheeler in [10]. In addition to its use in cipher-feedback mode, Wheeler also suggested WAKE as suitable for the production of a pseudo-random sequence for use as the keystream for a stream cipher. By changing the state machine from cipher-feedback to output-feedback mode the potential susceptibility to a chosen-plaintext attack is eliminated. This mode, referred to in [3] as WAKE-OFB, is shown in Fig. 1a. In the diagram all registers and signal paths are w -bits wide, with $w = 32$ for efficiency on 32-bit processors. This topology in conjunction with the invertibility of the mixing function results in a reversible generator.

In [3] it was shown that by running the keystream generator in reverse a threefold increase in computational parallelism could be achieved, resulting in very high throughput on processor architectures having instruction-level parallelism while claiming identical security to the non-reversed version. We call this form WAKE-ROFB.

4 WAKE-OFB and WAKE-ROFB as shift registers

While not originally portrayed as such, we can re-draw WAKE-OFB as an equivalent word-wide shift register. This is shown in Fig. 1b, again with all registers and signal paths being w -bits wide. In this form the shift register must be stepped four times to produce the new output that the parallel state machine of Fig. 1a produces in just one step. This is demonstrated by the state sequence listed in table 1.

Fig. 1. Alternate implementations of 4-stage WAKE-OFB

	T0	T1	T2	T3	T4=T0'
R_A	R3	R0'	R1'	R2'	R3'
R_B	R2	R3	R0'	R1'	R2'
R_C	R1	R2	R3	R0'	R1'
R_D	R0	R1	R2	R3	R0'
M_{Out}	R0'	R1'	R2'	R3'	R0''
Output	No	No	No	Yes	No

Table 1. Contents of registers R_A through R_D for time intervals T0 to T4

If we sub-sample the output of the shift-register version by four we produce the exact same output sequence as the original parallel version produces on consecutive steps. The serial form is particularly attractive for hardware implementation because, apart from there being just one mixing function to implement, the control logic for this topology is quite trivial. In hardware the serial form produces an average throughput of 8 bits per clock cycle which is still quite fast enough for most applications.

Note that in software *there is no distinction between the parallel and sub-sampled serial forms*. They are simply different but equally valid representations of a single computational flowgraph.

4.1 More registers, more iterations

For WAKE and its derivatives, adding more stages has been suggested as a convenient means of gaining extra security. Interpreting this strategy in the context of the serial form of WAKE-OFB we can see it as equivalent to increasing both the length of the shift register and the sub-sampling factor. i.e. 5-stage WAKE-OFB is equivalent to a five-stage word-wide shift register whose output is taken on every fifth step.

Now it becomes apparent that there are actually *two* degrees of freedom to be exploited. On the one hand we can increase the number of registers in the chain and independently we can select how many times to step the generator between outputs. Clearly, if we exercise these freedoms independently then the equivalent parallel hardware implementation, if we trouble ourselves to discover it, will be something other than the simple form in Fig. 1a. However this does not concern us since for hardware efficiency we will implement the serial form and this form also fully specifies the task in software.

Every time the shift register of Fig. 1b is stepped, the data in $\mathbf{R}_A$ participates in another iteration of the mixing function. If the mixing function is cryptographically useful then we can expect each additional iteration to contribute cryptographic strength to the contents of $\mathbf{R}_A$ in the same way that strength accumulates with the number of rounds in an iterated block cipher. Since the other registers are simply older copies of what was once one in $\mathbf{R}_A$, it does not matter which register is tapped for output from the viewpoint of cryptographic strength. Wherever the output is taken from it can reasonably be argued that the cryptographic strength generally increases with the sub-sampling factor.

Increasing the number of registers increases the amount of state that an attacker needs to deduce. It also increases the number of iterations of the mixing function by which the two inputs to the mixing function differ. In general, the greater this cryptographic separation the less chance there is of an unfortunate interaction between the two inputs, such as constructive reinforcement of some statistical bias. However, by analogy to differential characteristics in iterated block ciphers, if the mixing function is known to have an exploitable m -round characteristic, then it may be appropriate to avoid there being a small multiple of this number of register stages between the inputs of the mixing function.

We suggest that for a shift register with n -stages, as shown in Fig. 2a, it is ill advised to sub-sample by a factor of $n - 1$ since in this case the output stream reveals *both* inputs to the mixing function for some time steps, which may ease cryptanalysis. Also, any sub-sampling factor less than n will result in the contents of more than one register being known for at least some time steps, which diminishes one of the purposes for having more registers – i.e. increasing the amount of state that an attacker must deduce.

Fig. 2. Forward and reversed forms of generators with an arbitrary number of stages

4.2 Improving performance by running backwards

Cryptographic security of a pseudo-random sequence, as used for a stream cipher, demands that no one part of the sequence can be predicted from any other part of the sequence. This security property makes no distinction between the forward and time reversed versions of the pseudo-random sequence. Thus, if a *reversible* pseudo-random number generator, such as WAKE-OFB, produces a cryptographically secure sequence then that same generator running in reverse must also produce a cryptographically secure sequence.

In [3] it was observed that running the WAKE-OFB generator backwards exposed three times as much computational parallelism as was present in the forward version. We call the reversed version WAKE-ROFB. Since the forward and reversed versions produce the same keystream – simply in the opposite order from one another, we claim that their security is identical.

Just as we were able to draw an equivalent serial form for WAKE-OFB, we are able to draw an equivalent serial form for WAKE-ROFB. In Fig. 2a we

show the serial form of WAKE-OFB with the number of registers generalized¹. Fig. 2b shows the serial equivalent of WAKE-ROFB which is, not surprisingly, the time-reversed version of Fig. 2a. To get the time-reversed version we simply reverse the direction of data through each register and patch up the flowgraph accordingly. As part of this exercise the mixing function gets replaced by its inverse, $\mathbf{M}'$.

Now the reason for the increase in computational parallelism in WAKE-ROFB becomes obvious. If $\mathbf{R}_1$ through $\mathbf{R}_n$ in Fig. 2b are known then clearly the next $n - 1$ pairs of inputs to the mixing function are already determined, so in principle the results of these $n - 1$ mixing function evaluations can all be performed in parallel.

This provides us with another criterion to guide our choice of the number of register stages – it should be sufficient to let us effectively exploit the available instruction-level parallelism in the processors on which the cipher will run.

Previously, the available parallelism has been shown to be advantageous for efficiently exploiting instruction-level parallelism in software implementations, as reported in [3]. A notable advantage that this parallelism affords *hardware* implementations is the ability to pipeline the mixing function and thereby increase the maximum clock rate at which such implementations can operate.

Pipelining is a technique commonly used in hardware design as a means for increasing the maximum attainable clock rate of a system. The maximum clock rate of a system is bounded by the longest propagation path through combinational logic between any two registers. If a circuit topology allows additional registers to be placed in these long paths then the attainable clock rate can be increased.

In Fig. 2b we observe that all but one of the w -bit registers, i.e. $\mathbf{R}_1$ through $\mathbf{R}_{n-1}$, appear in series with the output of the mixing function. Thus, the mixing function can incorporate $n - 1$ pipeline stages while maintaining the exact same pseudo-random sequence as the original circuit. This is re-drawn for clarity in Fig. 2c from which it is clear that evaluation of the pipelined mixing function can take as long as $n - 1$ cycles so long as new inputs are accepted and a new output is produced, on every cycle. Of course in practice the registers would not simply be placed in series with the output of the mixing function, but instead would be buried inside. The optimum positioning of these registers when moved inside the mixing function is dependent on the specific propagation delays through its various paths and can be determined by standard hardware design techniques. We comment that speed-optimized positioning of the pipeline registers may place them at computational wavefronts crossed by more than w signals, each of which must be registered. So, the increase in speed due to pipelining may come at the cost of some small increase in register bits.

¹ The output tap has also been moved to the bottom to aid comparison with the other figures. This does not affect the function of the WAKE-OFB model, it simply advances the availability of the output.

Fig. 3 a) WAKE's Mixing function

b) Inverse mixing function

4.3 Trade-off between speed, security, and hardware efficiency

From the preceding discussions we see that there is a three-way trade-off between hardware efficiency, security, and speed (both software speed and hardware speed, with software speed being more of a concern).

The arrangement of Fig. 2a (WAKE-OFB) offers no cryptographic or performance advantages over that of Fig. 2b (WAKE-ROFB), and consequently has no reason to be used. The fundamental distinction between the two is that WAKE-ROFB allows multiple mixing functions to be evaluated in parallel while WAKE-OFB does not. This distinction is analogous to the difference between the Fibonacci and Galois forms of a linear feedback shift register.

In selecting the number of w -bit registers forming the shift register we trade hardware efficiency and software state against the amount of available parallelism (which up to a point translates into speed). For efficient operation on some legacy processors, notably the 'x86 family, it is desirable to keep the amount of state small so that it can stay resident in the very limited register set of these processors. This can be less of a constraint if efficient operation is only required on more recent family members like the Pentium II, since in them the MMX registers can relieve pressure on the legacy registers, even if the algorithm doesn't need the specialized MMX instructions.

In selecting the number of times to step the generator between outputs we directly trade speed against security. Indirectly we are also trading them both against hardware complexity, since in general we might expect to need fewer iterations for the same level of security if the mixing function is made more complex.

5 Design of the mixing function

WAKE's mixing function, $M(x, y, T)$, combines two 32-bit inputs, x and y , into one 32-bit output with the aid of a key-dependent 256×32 -bit look-up-table, T . After summing the two input words the low byte becomes the index into the table. The other three bytes are shifted down by one byte and then XORed with the table's output. By constraining the values in the upper byte lane of the otherwise 'random' entries of T to form a permutation of the numbers 0 to 255, the mixing function is made invertible in the sense that knowledge of the output word and one of the two input words is sufficient to uniquely specify the other input word. The action of the table-look-up is to permute the bottom byte and place it in the top byte while also providing confusion for the other three bytes. The right-shift serves to propagate changes to the right while the addition operation provides propagation to the left. The mixing function, and its inverse, are shown in Fig. 3.

An attraction of this mixing function is that it is very fast in software. Unfortunately, for our purposes the 256×32 -bit look-up-table presents an unacceptable hardware burden. Even if we count each bit as equivalent to just a single gate the table will take over 8000 gates. Our budget demands that we be an order of magnitude smaller than this.

5.1 Splitting the table

To address the hardware issue we modify the mixing function so that the look-up-table part may be compatibly implemented by either of two constructions. The first construction is as two 16×16 -bit look-up-tables, while the second is as a single 256×32 -bit table. The first construction is preferred for economic hardware implementation, while the second provides the fastest software implementation.

In the hardware case with two 16×16 -bit tables, half of the original address bits go to one table while the second table receives the others. The pair of 16-bit outputs are combined to produce a 32-bit word by interleaving in a fashion that is discussed later.

This strategy results in a construction for which *there is always an equivalent 256×32 -bit table* with a constrained form. Consequently *in software* we can continue to implement the mixing function by using a single 256×32 -bit table, thereby avoiding degrading the speed of mixing function evaluation.

The resultant [inverse] mixing function, for use with the register topology of Fig. 2b, is shown in Fig. 4. Compared to the inverse mixing function of Fig. 3b we have also changed the shift direction to be that of Fig. 3a and adopted the corresponding table address field and arithmetic operation. This has no cryptographic significance nor impact on the hardware efficiency but there is a benefit in software to extracting the low byte as the table address rather than the high byte. Low byte extraction is simply a masking operation, which employs the processor's ALU, while high byte extraction needs a shift. In processors with multiple execution units it is common to have more ALUs than shifters. Since we

Fig. 4. Modified mixing function with split table

also need to shift the other three bytes we want to avoid creating a bottleneck here.

In order to maintain the reversibility of the generator we keep the mixing function invertible by applying Wheeler’s trick to the upper nibble of each table, i.e. the upper nibble of each table contains a key-dependent pseudo-random permutation of that table’s four-bit address input.

While the remaining contents of the tables could simply be pseudo-random entries, as in WAKE’s original table, the table sizes in the new version are especially small so that statistical bias becomes more of a concern. To mitigate this concern we choose to fill each of the other three nibble lanes in both tables with their own key-dependent pseudo-random permutations.

5.2 Partitioning the address and re-combining the table outputs

A randomly filled 256-entry look-up-table will, with high probability, have outputs whose non-linear order is 8. However, outputs from a 16-entry table cannot have non-linear order greater than 4. In order to as rapidly as possible regain the non-linear complexity of the mixing function based on a single randomly filled 256-entry look-up-table, while incurring no additional hardware burden, we interleave the outputs of the two tables on a bit-by-bit basis, except that at byte boundaries the interleaving order is reversed.

This is illustrated below where bits $A_0 - A_{15}$ come from one table and $B_0 - B_{15}$ come from the other table:

$A_{15}B_{15}A_{14}B_{14}A_{13}B_{13}A_{12}B_{12}$ $B_{11}A_{11}B_{10}A_{10}B_9A_9B_8A_8$ $A_7B_7A_6B_6A_5B_5A_4B_4$ $B_3A_3B_2A_2B_1A_1B_0A_0$

Interleaving the table outputs on a bit-by-bit basis creates the maximum possible interaction between the two groups of bits in their next encounter with an adder, since any carries generated inside the adder then act to mix bits that came from independent tables. The deviation from strict bit-by-bit interleaving is so that the XOR stage always combines output bits from one table with those bits in the shifted path that have highest correlation with the other table (from an earlier mixing function). This again is to make all resulting bits as strongly dependent on both tables as possible.

We partition the original byte address into high and low nibbles for addressing the two smaller tables. This gives each table address a contribution from both tables in earlier operations, due to the interleaving of table outputs in a previous mixing function. The low nibble, which includes bit 0 that can have had no influence from table B through the action of a carry arising from the addition operation in the preceding mixing function, is used as the address for table B . Meanwhile the high nibble become the address for table A . In this way each table's address is composed of a roughly even contribution from both tables in earlier operations. Of the two groups of address bits, the one containing bit 0 has a greater bias towards table A , and by using bit 0 in table B 's address we help to even out this bias.

Our partitioning of the address bits has no influence on the performance of a software implementation since however we choose to partition them we can always construct an equivalent 256×32 -bit table.

5.3 Using a Feistel round as the mixing function

So far, reversibility of the word-wide shift register topology has been achieved by using a mixing function having as its non-linear element a pseudo-random permutation, thereby making the mixing function invertible. However, such a construction is not strictly necessary to ensure that the state machine is reversible.

In Fig. 5a we illustrate a more general form of the mixing function which guarantees the reversibility of the state machine. $\mathbf{P}_x$ and $\mathbf{P}_z$ are pseudo-random *permutations* while $\mathbf{F}_y$ is a pseudo-random *function*. While Fig. 5a shows $\mathbf{P}_z$ and $\mathbf{F}_y$ being combined by exclusive-OR, the only requirement on the combining operation is that it be invertible, so for instance addition modulo 2^w is equally suitable. Cryptographically useful mixing functions require at least one out of $\mathbf{P}_x$, $\mathbf{P}_z$, and $\mathbf{F}_y$ to be non-trivial.

We can now see that the earlier proposed mixing function is of the form having non-trivial $\mathbf{P}_z$, while $\mathbf{P}_x$ and $\mathbf{F}_y$ are identity operators (i.e. absent), with addition used as the combining operation.

If we consider the case where only $\mathbf{F}_y$ is present, as shown in Fig. 5b, we note that in essence the mixing function is then equivalent to the round function of

a) with generalized mixing function b) with 'Feistel round' as mixing function

Fig. 5. Reversible generators with alternative types of mixing function

a Feistel network. This raises the interesting possibility to apply the techniques from block cipher design and analysis to this topology, knowing that the resulting stream cipher is guaranteed to have a high degree of parallelism. Of course, F_y is not required to be a *fixed* function, but could instead vary by shift-register step in the same way that common practice in Feistel-based block cipher design is for the key schedule to introduce some variation in the round function by round number.

6 Key scheduling and stream re-synchronization

For a complete stream cipher definition two further components – initialization of the key-dependent look-up-tables (key scheduling), and stream re-synchronization, need to be specified. We discuss these two operations in light of our goal of hardware efficiency.

We anticipate that even in applications where encryption is performed in hardware there will commonly exist a processor of some kind, such as a micro-controller used to orchestrate the operation of a consumer electronics device. While such a CPU cannot be relied upon to perform ciphering of high speed data streams, it is not unreasonable to call upon it to assist with initializing a hardware-based mixing function's key-dependent look-up-tables, so long as key changes are relatively infrequent. For this reason, we allow the key scheduling

operation to be rather more complicated than would be desirable if it had to be performed strictly in hardware, instead concerning ourselves only to make it relatively easy to implement even on a low-end microcontroller. Specifically, only the construction of the pair of 16×16 -bit tables need be straightforward for a microcontroller, not construction of the equivalent 256×32 -bit table, since the large table is only required for efficient *software-only* cipher implementations.

On the other hand, we want to allow for the possibility of frequent stream re-synchronization without concern for unduly burdening the microcontroller in the case that encryption is being performed in hardware. For this reason we keep the re-synchronization process very simple so that its implementation in hardware alone is trivial. We perform re-synchronization by setting the shift register to a new state that is solely dependent on a supplied initialization vector and then step the generator until its state is thoroughly dependent on the contents of the look-up-tables, discarding the generator's output along the way.

7 Suggested configurations

To provide the amount of parallelism necessary to fully exploit a processor capable of performing four or five 32-bit operations per cycle we need five or six 32-bit register stages in our shift register. This is also an acceptable number for an efficient register-based implementation on a Pentium II.

A hardware implementation of this stream cipher, using the proposed split-table mixing function with a 32-bit $\times$ 5-stage shift register and the necessary control logic, takes about 2000 gates (with key scheduling assumed to be done in software).

We suggest two models:

- **5-stage shift register with the output sampled on every 8th step.**

This model's 5-stage shift register allows four mixing functions to be evaluated concurrently, which lets it effectively exploit the instruction-level parallelism in media-processors such as the TriMedia TM-1000. By increasing the number of mixing function iterations between outputs to eight from WAKE-ROFB's original four we hope to amply compensate for the split look-up-table.

Optimized implementations of this model achieve a throughput of 340 Mbps on a 266 MHz Pentium II, and 270 Mbps on a 100 MHz TriMedia chip, or 1.3 and 2.7 bits per cycle respectively. While the TriMedia performance can be achieved with good C-code, the reported Pentium II performance requires assembly coding. In particular, with the Pentium II's MMX registers being used to hold the 5-stage shift register and the standard registers being used for memory pointers and the loop counter, it is possible to avoid any redundant register-to-register moves or any memory accesses other than the data buffer being encrypted.

In hardware, shift-register cycle times of under 20 ns are easily achievable by pipelining the mixing function. This gives a 200 Mbps throughput at a 50 MHz clock rate.

In the Appendix we provide specimen C-code for this model, which we refer to as WWNFSR-5/8. The Appendix includes illustrative routines for key scheduling and re-synchronization operations using a 128-bit key and 64-bit initialization vector. Re-synchronization of the form specified takes about as long as ciphering 32 bytes. Key scheduling software for hardware-assisted implementations need only perform the first half of the key scheduling routine in which the pair of 16×16 -bit tables are constructed.

- **4-stage shift register with the output sampled on every 4th step.**

This model is equivalent to 4-stage WAKE-ROFB except that with our modified mixing function we expect it to be weaker. Four is the minimum number of iterations necessary for complete diffusion of the mixing function.

While this model only has three-fold parallelism this has not been found to be the limiting factor on the Pentium II. To get greater than two-fold parallelism out of the Pentium II requires taking advantage of the SIMD parallelism available through the MMX instructions. However, in practice we have not been able to take effective advantage of these because of our use of table-look-ups. The MMX instructions have no support for table-look-up, with the result that the benefit afforded by doing logical or arithmetic operations on two 32-bit words at a time (by using MMX) is negated by the inefficiency of re-arranging the data into the non-MMX registers from which table-look-ups can be performed. Similarly, the reported performance for WWNFSR-5/8 takes no advantage of the MMX instructions' SIMD parallelism, but simply uses the 64-bit MMX registers as 32-bit registers with the upper 32-bits going to waste.

Optimized implementations of this model achieve a throughput in excess of 600 Mbps on a 266 MHz Pentium II, and 340 Mbps on a 100 MHz TriMedia chip, or 2.3 and 3.4 bits per cycle respectively. In hardware it has a throughput of 400 Mbps at a 50 MHz clock rate.

The design intent of these non-linear finite state machines is that they behave as pseudo-random permutation generators. If such is the case then we expect the cycle length of a w -bit $\times$ n -stage version to be, on average, 2^{wn-1} . Cycle lengths have been determined experimentally for some artificially small models and found to be as expected. This is encouraging for the full-sized models because small models have less ability to hide statistical biases arising from 'boundary effects'. Statistical tests on full sized models have not revealed any weaknesses in these generators, however this alone should not be taken as an indication of their strength.

8 Conclusion

We have presented the word-wide non-linear-feedback shift register as a topology that offers a high degree of parallelism together with the ability to trade speed versus security by considering it as an iterated structure.

The topology can be used with any invertible non-linear mixing function, including the round function of a Feistel network. This represents fertile ground for future research – to identify suitable functions that maximize the security for a given amount of resources, and suggests the possibility that techniques developed for analyzing block ciphers may also be applicable to this topology.

The inherent parallelism present in the shift-register topology indicates that a stream cipher built this way is likely to be substantially faster than a block cipher used in OFB mode, even when both ciphers use the same round function and number of rounds/iterations. It remains to be demonstrated in this case that the speed increase does not in some way compromise security.

We suggest a non-linear mixing function derived from that of WAKE, but with the look-up-table split in two, thereby substantially reducing the hardware requirements. The potential loss of strength can be made up by increasing the number of iterations between outputs, albeit at a corresponding modest reduction in speed. We consider this to be an acceptable trade-off for the hardware efficiency gained.

References

1. R. J. Anderson, “On Fibonacci Keystream Generators”, *Fast Software Encryption* (Ed. B. Preneel), LNCS 1008, Springer-Verlag, 1995, pp. 346-352
2. U. Blöcher, M. Dichtl, “Fish: A Fast Software Stream Cipher”, *Fast Software Encryption* (Ed. R. Anderson), LNCS 809, Springer-Verlag, 1994, pp. 41-44
3. C. S. K. Clapp, “Optimizing a Fast Stream Cipher for VLIW, SIMD, and Superscalar Processors”, *Fast Software Encryption* (Ed. E. Biham), LNCS 1267, Springer-Verlag, 1997, pp. 273-287
4. J. Daemen, *Cipher and Hash Function Design - strategies based on linear and differential cryptanalysis*, Doctoral Dissertation, K.U. Leuven, March 1995, pp. 172-180
5. D. Knuth, *The Art of Computer Programming, Volume 2 - Seminumerical Algorithms*, Second edition, Addison Wesley, 1981
6. D. Coppersmith, H. Krawczyk, Y. Mansour, “The Shrinking Generator”, *Advances in Cryptology - Crypto '93*, LNCS 773, Springer-Verlag, 1994, pp. 22-39
7. P. Rogaway and D. Coppersmith, “A Software-Optimized Encryption Algorithm”, *Fast Software Encryption* (Ed. R. Anderson), LNCS 809, Springer-Verlag, 1994, pp. 56-63
8. B. Schneier, *Applied Cryptography, Second Edition*, John Wiley & Sons, 1996
9. B. Schneier, “Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)”, *Fast Software Encryption* (Ed. R. Anderson), LNCS 809, Springer-Verlag, 1994, pp. 191-204
10. D. J. Wheeler, “A Bulk Data Encryption Algorithm”, *Fast Software Encryption* (Ed. R. Anderson), LNCS 809, Springer-Verlag, 1994, pp. 127-134

Appendix A. WWNFSR-5/8 Reference Implementation

The following C-code implements the recommended 5-stage/8-step word-wide non-linear-feedback shift register model. Specimen routines for key scheduling and stream re-synchronization are included for the purposes of providing a complete cipher proposal for cryptanalytic investigation. This code is a functional reference only and does not represent all the performance optimizations embodied in the benchmarked code.

```

/* WWNFSR-5/8, C-code reference, version 1.0 */
/* 5-stage word-wide shift register with output sampled on every 8th step. */
/* Key-schedule illustrates mixing-function with split look-up-table. */
typedef unsigned long UINT32; /* 32-bit unsigned integer */
/* Array sizes - UINT32 T[256], key[4], IVec[2], state[5]; */

/* addition is modulo 232, >> is right shift with zero fill */
#define M(y,z,T) ((y) + (((z) >> 8) ^ T[(z) & 0xff]))

void ofb_crypt(UINT32 *In, UINT32 *Out, int length, UINT32 *T, UINT32 *state)
{
    UINT32 R1, R2, R3, R4, R5, Rt;
    int i;

    R1=state[0]; R2=state[1]; R3=state[2]; R4=state[3]; R5=state[4];
    for (i = 0; i < length; i++)
    {
        /* Naive implementation of shift-register obscures available parallelism */
        /* for (shiftstep = 0; shiftstep < 8; shiftstep++) */
        /* { Rt = M(R4,R5,T); R5=R4; R4=R3; R3=R2; R2=R1; R1=Rt; } */

        /* Logically equivalent flattened form makes four-fold parallelism clear */

        Rt = M(R2,R3,T); /*
        R3 = M(R3,R4,T); /* All four mixing functions
        R4 = M(R4,R5,T); /* can be evaluated in parallel
        R5 = M(R1,R2,T); /*

        R2 = M(Rt,R3,T); /*
        R3 = M(R3,R4,T); /* All four mixing functions
        R4 = M(R4,R1,T); /* can be evaluated in parallel
        R1 = M(R5,Rt,T); /*

        Out[i] = In[i] ^ R5; /* Execution can overlap with second
        /* group of mixing functions */
    }

    state[0]=R1; state[1]=R2; state[2]=R3; state[3]=R4; state[4]=R5;
}

void resync(UINT32 *IVec, UINT32 *T, UINT32 *state)
/* set state from initialization vector and discard first eight words */
{
    UINT32 temp[8]; /* bit-bucket */

    state[0] = state[3] = state[4] = IVec[0];
    state[1] = state[2] = IVec[1];

    ofb_crypt(temp, temp, 8, T, state);
}

```

```

void key_sched(UINT32 *key, UINT32 *T)
/* build 16 x 16 look-up-tables Tb, Ta, and 256 x 32 table T from 128-bit key */
{
    UINT32 i, j, k, n, mask, xordif;
    UINT32 TbTa[16]; /* pair of 16 x 16 look-up-tables stacked side-by-side */
    /* initialize permutations in each nibble-lane of TbTa */
    for (i = j = 0; i < 16; i++) { TbTa[i] = key[3] ^ j; j = j + 0x11111111; }
    k = 0; mask = 0x00f0f0ff; /* mask selects four out of eight nibbles */
    for (j = 0; j < 8; j++) /* do for each of 8 mask values */
    {
        for (i = 0; i < 16; i++) /* do for each of 16 table entries */
        {
            /* scan across key and table nibbles to define table entry for swap */
            n = (TbTa[i] >> (j*4)) + (key[(j*2 + i/8) & 0x3] >> ((i*4) & 0x1c));
            k = (k + n) & 0x0f;

            /* swap masked nibbles between TbTa[i] and TbTa[k] */
            xordif = (TbTa[i] ^ TbTa[k]) & mask;
            TbTa[i] = TbTa[i] ^ xordif;
            TbTa[k] = TbTa[k] ^ xordif;
        }
        mask = (mask << 4) | (mask >> 28); /* rotate mask left by 4 bits */
    }
    /* build 256 x 32 table T by interleaving left and right halves of TbTa */
    UINT32 a, b;
    UINT32 expand[16] = { 0x00,0x01,0x04,0x05, 0x10,0x11,0x14,0x15,
                          0x40,0x41,0x44,0x45, 0x50,0x51,0x54,0x55 };
    for (i = 0; i < 256; i++) { T[i] = 0; } /* clear look-up-table T */
    for (j = 0; j < 16; j++)
    {
        k = TbTa[j];
        a = (expand[(k >> 0) & 0xf] << 0) ^ (expand[(k >> 4) & 0xf] << 9)
            ^ (expand[(k >> 8) & 0xf] << 16) ^ (expand[(k >> 12) & 0xf] << 25);
        b = (expand[(k >> 16) & 0xf] << 1) ^ (expand[(k >> 20) & 0xf] << 8)
            ^ (expand[(k >> 24) & 0xf] << 17) ^ (expand[(k >> 28) & 0xf] << 24);
        for (i = 0; i < 16; i++) /* fill look-up-table T */
            { T[i+16*j] = T[i+16*j] ^ a; T[16*i+j] = T[16*i+j] ^ b; }
    } } }

```

Appendix B. Test Case

```

void test(void)
{
    UINT32 key[4] = { 0x12345678, 0x98765432, 0xabcdef01, 0x10fedcba };
    UINT32 IVec[2] = { 0xbabeface, 0xf0eld2c3 }; /* Initialization Vector */
    UINT32 text[4] = { 0x1234abcd, 0xa0b1c2d3, 0x1a2b3c4d, 0x55667788 };
    UINT32 T[256], state[5];
    int i;

    key_sched(key, T); /* Schedule key */
    resync(IVec, T, state); /* Initialize generator state using IV */
    for (i = 0; i < 256; i++) /* Encrypt text buffer 256 times */
        { ofb_crypt(text, text, 4, T, state); }
    for (i = 0; i < 4; i++) { printf("0x%08lx ", text[i]); } printf("\n");
}

/* final text[] == { 0xe5650b3d, 0xfdb4dca1, 0xc904b128, 0xd25f1934 } */

```