Phelix

Fast Encryption and Authentication in a Single Cryptographic Primitive

Doug Whiting[1], Bruce Schneier[2], Stefan Lucks[3], and Frédéric Muller[4]

[1] HiFn, dwhiting@hifn.com
[2] Counterpane Internet Security, schneier@counterpane.com
[3] Universität Mannheim, lucks@th.informatik.uni-mannheim.de
[4] DCSSI Crypto Lab, frederic.muller@sgdn.pm.gouv.fr

Abstract. Phelix[1] is a high-speed stream cipher with a built-in MAC functionality. It is efficient in both hardware and software. On current Pentium CPUs, Phelix has a per-packet overhead of less than 900 clocks, plus a per-byte cost well under 8 clocks per byte, comparing very favorably with the best AES (encryption-only) implementations, even for small packets.

Keywords: Stream cipher, MAC, authentication, encryption.

1 Introduction

Securing data in transmission is the most common real-life cryptographic problem. Basic security services require both encryption and authentication. This is almost always done using a symmetric cipher—public-key systems are only used to set up symmetric keys—and a Message Authentication Code (MAC).

The AES process provided a number of very good block cipher designs, as well as a new block cipher standard. The cryptographic community learned a lot during the selection process about the engineering criteria for a good cipher. AES candidates were compared in performance and cost in many different implementation settings. We learned more about the importance of fast rekeying and tiny-memory implementations, the cost of S-boxes and circuit-depth for hardware implementations, the slowness of multiplication on some platforms, and other performance considerations.

The community also learned about the differences between cryptanalysis in theory and cryptanalysis in practice. Many block cipher modes restrict the types of attack that can be performed on the underlying block cipher. Yet the generally accepted attack model for block ciphers is very liberal. Any method that distinguishes the block cipher from a random permutation is considered an attack. Each block cipher operation must protect against all types of attack. The resulting overengineering leads to inefficiencies.

Computer network properties like synchronization and error correction have eliminated the traditional synchronization problems of stream-cipher modes like OFB. Furthermore, stream ciphers have different implementation properties that restrict the cryptanalyst. They only receive their inputs once (a key and a nonce) and then produce a long stream of pseudorandom data. A

[1] Pronounced "felix" (rhymes with "helix").

stream cipher can start with a strong cryptographic operation to thoroughly mix the key and nonce into a state, and then use that state and a simpler mixing operation to produce the keystream. If the attacker tries to manipulate the inputs to the cipher he encounters the strong cryptographic operation. Alternatively, he can analyze the keystream, but this is a static analysis only. As far as we know, static attacks are much less powerful than dynamic attacks. As there are fewer cryptographic requirements to fulfill, we believe that the keystream generation function can be made significantly faster, per message byte, than a block cipher can be. Given the suitability of stream ciphers for many practical tasks and the potential for faster implementations, we believe that stream ciphers are a fruitful area of research.

Additionally, a stream cipher is often implemented—and from a cryptographic point of view, should always be implemented—together with a MAC. Encryption and authentication go hand in hand, and significant vulnerabilities can result if encryption is implemented without authentication. Outside the cryptographic literature, not using a proper MAC is one of the commonly encountered errors in stream cipher systems. A stream cipher with a built-in MAC is much more likely to be used correctly, because it provides a MAC without the associated performance penalties.

Phelix is an attempt to combine all these lessons. It is closely related to a prior cipher, Helix [Helix03], proposed by some of the same authors.

2 An Overview of Phelix

Phelix is a combined stream cipher and MAC function, and directly provides authenticated encryption functionality. By incorporating the plaintext into the stream cipher state, Phelix can provide authentication functionality without extra cost [Gol00].

The design strength of Phelix is 128 bits, which means we expect that no attack on the cipher exists that requires fewer than 2^{128} Phelix block function evaluations to be carried out. Phelix can process data in less than 7 clock cycles per byte on a Pentium M CPU, more than twice as fast as the best known AES implementation.

Phelix uses a 256-bit key and a 128-bit nonce. The key is secret, and the nonce is typically public knowledge. Phelix is optimized for 32-bit platforms; all operations are on 32-bit words. The only operations used are addition modulo 2^{32}, exclusive or, and rotation by fixed numbers of bits. The design philosophy of Phelix can be summarized as "many simple rounds."

Phelix has a state that consists of nine words of 32 bits each. The state is broken up into two groups: 5 "active" state words, which participate in the block update function, and 4 "old" state words that are only used in the keystream output function. A single round of Phelix consists of adding (or XORing) one active state word into the next, and rotating the first word. This is shown in Figure 1, where the active state words are shown as vertical lines. Multiple rounds are applied in a cyclical pattern to the active state. The horizontal lines of the rounds wind themselves in helical fashion through the five active state words. Twenty rounds make up one block (see Figure 2).

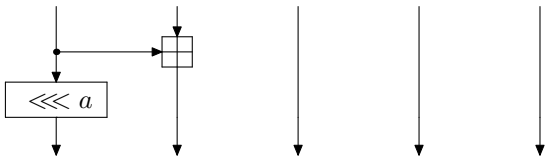

Fig. 1. A single round of Phelix

Phelix actually uses two intertwined helices; a single block contains two full turns of each of the helices. The name Phelix was derived by a contraction of sorts, from the fact that there are five "strands" in the helices, so that the structure can be thought of as a penta-helix.

During each block, several other activities occur. During block i, one word of keystream is generated (S_i), two words of key material are added ($X_{i,0}$ and $X_{i,1}$), and one word of plaintext is added (P_i). The output state of one block is used as input to the next, so the computations shown in figure 2 are all that is required to process 4 bytes of the message. As with any stream cipher, the ciphertext is created by XORing the plaintext with the keystream (not shown in the figure).

At the start of an encryption, a starting state is derived from the key and nonce. The key words $X_{i,j}$ depend on the key, the length of the input key, the nonce, and the block number i. State-guessing attacks are made more difficult by adding key material at double the rate at which key stream material is extracted. At the end of the message, some extra processing is done, after which a 128-bit MAC tag is produced to authenticate the message.

The structure of Phelix is very similar to that of Helix [Helix03]. In fact, the block functions of the two ciphers are identical, except for the keystream output function. The enlarged internal state of Phelix, along with moving the keystream output point to force larger plaintext diffusion, would seem to increase the security margin significantly. These changes were made largely in response to [Mul04].

3 Definition of Phelix

The Phelix encryption function takes as input a variable-length key U of up to 256 bits (32 bytes), a 128-bit (16-byte) nonce N, and a plaintext P. It produces a ciphertext message and a tag that provides authentication. The decryption function takes the key, nonce, ciphertext, and tag, and produces either the plaintext message or an error if the authentication failed.

3.1 Preliminaries

Phelix operates on 32-bit words, while the inputs and outputs are 8-bit bytes. In all situations, Phelix uses the least-significant-byte-first convention. A sequence of bytes x_i is identified with a sequence of words X_j by the relations

$$X_j := \sum_{k=0}^{3} x_{(4j+k)} \cdot 2^{8k} \qquad\qquad x_i := \left\lfloor \frac{X_{\lfloor i/4 \rfloor}}{2^{8(i \bmod 4)}} \right\rfloor \bmod 2^8$$

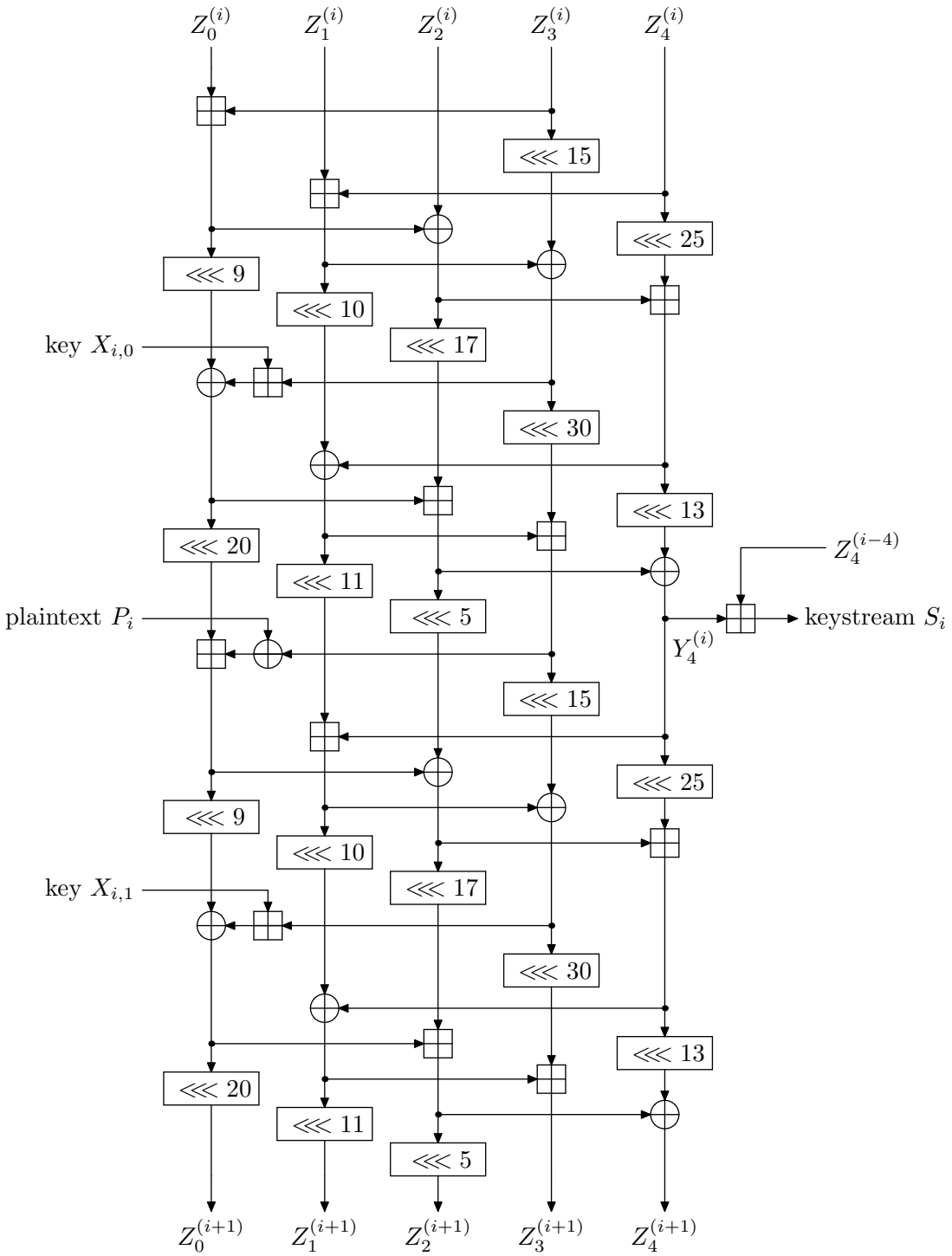

Fig. 2. One block of Phelix encryption

These two equations are complementary and show the conversion both ways.

Let $\ell(x)$ denote the length of a string of bytes x. The input key U consists of a sequence of bytes $u_0, u_1, \ldots, u_{\ell(U)-1}$ with $0 \leq \ell(U) \leq 32$. The key is processed through the key mixing function, defined in section 3.7, to produce the working key which consists of 8 words $K_0, \ldots, K_7$.

The nonce N consists of 16 bytes, interpreted as 4 words $N_0, \ldots, N_3$.

The plaintext P and ciphertext C are both sequences of bytes of the same length, with the restriction that $0 \leq \ell(P) < 2^{64}$. Both are manipulated as a sequence of words, P_i and C_i, respectively. The last word of the plaintext and ciphertext might be only partially used. The "extra" plaintext bytes in the last word are taken to be zero. The "extra" ciphertext bytes are irrelevant and never used. Note that the cipher is specified for zero-length plaintexts; in this case, no data is encrypted and only a MAC is generated.

3.2 A Block

Phelix consists of a sequence of blocks. The blocks are numbered sequentially, which assigns each block a unique number i. At the start of block i, the active state consists of five words: $Z_0^{(i)}, \ldots, Z_4^{(i)}$; at the end of the block, the active state consists of $Z_0^{(i+1)}, \ldots, Z_4^{(i+1)}$, which forms the input to the next block with number $i+1$. Block i also uses as input two key words $X_{i,0}$ and $X_{i,1}$, the plaintext word P_i, and a previous state word $Z_4^{(i-4)}$.

The complete block function is illustrated in figure 2. All values are 32-bit words; exclusive or is denoted by $\oplus$, addition modulo 2^{32} is denoted by $\boxplus$, and rotation by $\lll$. The block function actually consists of two applications of a "half-block" function H, defined as:

Function $H(w_0, w_1, w_2, w_3, w_4,\ K_0, K_1)$
Begin

$$
\begin{array}{ll}
w_0 := w_0 \boxplus (w_3 \oplus K_0); & w_3 := w_3 \lll 15; \\
w_1 := w_1 \boxplus\ w_4; & w_4 := w_4 \lll 25; \\
w_2 := w_2 \oplus\ w_0; & w_0 := w_0 \lll\ 9; \\
w_3 := w_3 \oplus\ w_1; & w_1 := w_1 \lll 10; \\
w_4 := w_4 \boxplus\ w_2; & w_2 := w_2 \lll 17; \\
 & \\
w_0 := w_0 \oplus (w_3 \boxplus K_1); & w_3 := w_3 \lll 30; \\
w_1 := w_1 \oplus\ w_4; & w_4 := w_4 \lll 13; \\
w_2 := w_2 \boxplus\ w_0; & w_0 := w_0 \lll 20; \\
w_3 := w_3 \boxplus\ w_1; & w_1 := w_1 \lll 11; \\
w_4 := w_4 \oplus\ w_2; & w_2 := w_2 \lll\ 5;
\end{array}
$$

Return $(w_0, w_1, w_2, w_3, w_4)$;
End.

Given the function H, the block function, shown in figure 2, is computed as follows:

$$
\begin{aligned}
(Y_0^{(i)}, Y_1^{(i)}, Y_2^{(i)}, Y_3^{(i)}, Y_4^{(i)}) &:= H(Z_0^{(i)}, Z_1^{(i)}, Z_2^{(i)}, Z_3^{(i)}, Z_4^{(i)}, \quad 0, X_{i,0}) \\
(Z_0^{(i+1)}, Z_1^{(i+1)}, Z_2^{(i+1)}, Z_3^{(i+1)}, Z_4^{(i+1)}) &:= H(Y_0^{(i)}, Y_1^{(i)}, Y_2^{(i)}, Y_3^{(i)}, Y_4^{(i)}, \quad P_i, X_{i,1})
\end{aligned}
$$

Each block produces one word of keystream $S_i := Y_4^{(i)} + Z_4^{(i-4)}$. The ciphertext words are defined by $C_i := P_i \oplus S_i$.

In the remainder of this paper, the terms "block," and "block function" are used interchangeably.

3.3 Key Words for Each Block

The expanded key words are derived from the working key $K_0, \ldots, K_7$, the nonce $N_0, \ldots, N_3$, the input key length $\ell(U)$, and the block number i. We first extend the nonce to 8 words by defining $N_k := (k \bmod 4) - N_{k-4} \pmod{2^{32}}$ for $k = 4, \ldots, 7$. The key words for block i are then defined by

$$\begin{aligned} X_{i,0} &:= K_{i \bmod 8} \\ X_{i,1} &:= K_{(i+4) \bmod 8} + N_{i \bmod 8} + X'_i + i + 8 \\ X'_i &:= \begin{cases} \lfloor (i+8)/2^{31} \rfloor & \text{if } i \bmod 4 = 3 \\ 4 \cdot \ell(U) & \text{if } i \bmod 4 = 1 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

where all additions are taken modulo 2^{32}. Note that X'_i encodes bits 31 to 62 of the value $i + 8$; this is not the same as the upper 32 bits of $i + 8$.

3.4 Initialization

A Phelix encryption is started by setting

$$\begin{aligned} Z_j^{(-8)} &:= K_{j+3} \oplus N_i && \text{for } j = 0, \ldots, 3 \\ Z_4^{(-8)} &:= K_7 \\ Z_4^{(i)} &:= 0 && \text{for } i = -12, \ldots, -9 \\ P_i &:= 0 && \text{for } i = -8, \ldots, -1 \end{aligned}$$

Eight blocks are then applied, using block number $i = -8, \ldots, -1$. For these blocks, the generated keystream words are discarded.

3.5 Encryption

After the initialization, the plaintext is encrypted. Let $k := \lfloor (\ell(P) + 3)/4 \rfloor$ be the number of words in the plaintext. The encryption consists of k blocks numbered 0 to $k - 1$. Each block generates one word of keystream, which is used to encrypt one word of the plaintext. Depending on $\ell(P) \bmod 4$, between 1 and 4 of the bytes of the last keystream word are used.

3.6 Computing the MAC

Just after the block that encrypted the last plaintext byte, one of the state words is modified. The internal state word $Z_0^{(k)}$ is XORed with the value 0x912d94f1.[2] Using this modified state, eight blocks, numbered $k, \ldots, k + 7$

[2] This constant is constructed by taking the six least significant bits of each of the ASCII characters of the string "Helix," and setting the bits before and after these 30 bits.

are applied for post-mixing. For these blocks, the plaintext word P_i is defined as $\ell(P) \bmod 4$, and the generated keystream is discarded . After the post-mixing, four more blocks, numbered $k+8, \ldots, k+11$, are applied, using the same plaintext input word (i.e., $\ell(P) \bmod 4$). The keystream words generated by these four blocks form the MAC tag.

3.7 Key mixing

The key mixing converts a variable-length input key U to the fixed-length working key, K.

First, the Phelix block function is used to create a round function R that maps 128 bits to 128 bits, as follows:

Function $R(w_0, w_1, w_2, w_3)$
Begin
  Local Variable $w_4 \quad := \ell(U) + 64;$
  $(w_0, w_1, w_2, w_3, w_4) := H(w_0, w_1, w_2, w_3, w_4, 0, 0);$
  $(w_0, w_1, w_2, w_3, w_4) := H(w_0, w_1, w_2, w_3, w_4, 0, 0);$
  Return (w_0, w_1, w_2, w_3);
End.

The input key U is first extended with $32 - \ell(U)$ zero bytes. The 32 key bytes are converted to 8 words $K_{32}, \ldots, K_{39}$. Further key words are defined by the equation

$$(K_{4i}, \ldots, K_{4i+3}) := R(K_{4i+4}, \ldots, K_{4i+7}) \oplus (K_{4i+8}, \ldots, K_{4i+11})$$

for $i = 7, \ldots, 0$. The words $K_0, \ldots, K_7$ form the working key of the cipher. (This recursion defines a Feistel-type cipher on 256-bit blocks.)

3.8 Decryption

Decryption is almost identical to encryption. The only differences are:

- The keystream S_i generated after the first application of the H function in each block is used to decrypt the ciphertext, producing the plaintext word that is used in the second application of the H function within the block. The implementation must insure that any unused bytes of the final plaintext word are taken as zero for purposes of computing the block function, regardless of value of the extra keystream bytes.
- Once the tag has been generated, it is compared to the tag provided. If the two values are not identical, all generated data (i.e., the keystream, plaintext, and tag) is destroyed.

4 Implementation

Compared to many ciphers, Phelix is relatively easy to implement in software. If 32-bit addition, exclusive or, and rotation functions are available, all the functions are easily implemented. Phelix is also fast. A single round takes only

one clock cycle to compute on most current Pentium CPUs, because the superscalar architecture can perform an addition or XOR simultaneously with a 32-bit rotation. A block of Phelix takes 20 cycles plus some overhead for the handling of the plaintext, keystream, and ciphertext. Our assembly language implementation requires less than 7 clock cycles per byte on a Pentium M CPU, and under 14 clocks per byte using optimized C (gcc version 3.2). This compares to about 14 clock cycles per byte for the best known AES implementation (in assembler) on the same platform. [3]

The per-packet processing overhead of Phelix is due mainly to the eight words of initialization and twelve words of MAC processing. To first order, this overhead can be thought of as processing an extra 20 words (80 bytes) per packet. In assembler on the Pentium M, there is also roughly an additional 200 clocks of general setup overhead (including nonce mixing into $X_{i,1}$), for a total of 750-850 clocks per packet. By comparison, this overhead is approximately the time required to process a single 64-byte block of SHA-1 or three 16-byte blocks of AES on the same platforms, but Phelix performs both encryption and MAC in the same operation.

Operation	Version	Packet Size (N)			Approximate Equation (clks)
		64 bytes	256 bytes	1024 bytes	
Encrypt	C	41.6 cpb	20.3 cpb	15.0 cpb	$1810 + 13.2\,N$
Decrypt	C	42.3 cpb	21.1 cpb	15.8 cpb	$1610 + 14.0\,N$
Encrypt	ASM	18.5 cpb	9.8 cpb	7.4 cpb	$810 + 6.6\,N$
Decrypt	ASM	18.2 cpb	9.6 cpb	7.4 cpb	$750 + 6.7\,N$

Empirical Phelix software speeds on a Pentium M CPU are given in the table above, where "cpb" indicates clocks per byte. These speeds are for an "all-in-one" call, where the entire packet is processed with a single call. The basic software model for Phelix is a slope-intercept equation, where the slope is under 7 clocks/byte and the intercept is about 800 clocks per packet, in assembler on a Pentium M. Minor speed differences occur due to encryption vs. decryption, and to various compilers and CPU versions, For example, on a Pentium III CPU, the slope is about 7.5 cpb, and the intercept is about 850 clocks. Using an "incremental" API, where each phase of the operation (e.g., nonce setup, data encryption/decryption, MAC output) is performed with a separate call, adds an extra per-packet overhead of about 300 CPU clocks in assembler and 100 clocks in C.

On a Pentium 4 CPU (P4), the same Phelix assembler code runs in about $1100+10N$ clocks per packet, and the C code runs in about $3100+26N$ clocks per packet. Both of these numbers are considerably slower than the Pentium M benchmarks. Perhaps recoding Phelix specifically for a P4 CPU could improve these numbers, but the P4 has a (well deserved) reputation for very high clock

[3] This is a somewhat unfair comparison. Most block cipher modes only provide encryption *or* authentication, so two passes over the message are required. The alternative is to use one of the new authenticated encryption modes, such as [Jut01], but they are all patented and require a license.

frequencies and relatively low instructions per clock, compared to the Pentium III and M CPUs.

The key mixing only needs to be done once for each key value and implements a Feistel cipher, so it can be done in place. The key mixing function takes about 400 clocks in assembler on a Pentium M, and under 600 clocks in C (gcc version 3.2). Within a packet, the $X_{i,1}$ key words can be mostly precomputed, with only the block number being added every block. Implementations that limit the plaintext size to 2^{32} bytes can ignore the upper bits of the block number in the definition of X'_i, because these bits will always be zero.

No algorithm setup is required for Phelix. There are no tables to be built. The code size on a Pentium is under 5K bytes for the optimized assembler version, including both all-in-one and incremental APIs, and under 6K bytes for a C version. The code size obviously depends on the amount of optimization and loop unrolling used. For example, the all-in-one assembler version could be coded in under 2K bytes at some cost in performance, although we have not implemented such a version.

Phelix is also fast in hardware. The rotations incur no gate delays, only wiring delays, although they do consume routing resources in chip layouts. The critical path through the block function consists of 6 additions and 5 XORs. As the critical path contains no rotations, a certain amount of ripple of the 32-bit adders can be overlapped, with the lower bits being produced and used before the upper bits are available. A more detailed analysis of this overlapping is required for any high-speed implementation. A conservative estimate for a relatively low-cost ASIC layout is 2.5 ns per 32-bit adder and 0.5 ns per XOR, which adds up to 17.5 ns/block. This translates to more than 200 MByte per second, or just under 2 Gbit per second. We roughly estimate that such a design would consume fewer than 20,000 gates. Such a design would incur about a 20 clock overhead per packet in order to process the initialization and MAC blocks.

We are not aware of any special efforts required to avoid implementation weaknesses, other than standard practices required for dealing with cryptographic primitives.

5 Use

One of the dangers of a stream cipher is that the keystream will be reused. To avoid this problem, Phelix imposes a few restrictions on the sender and receiver:

- The sender *must* ensure that each (K,N) pair is used at most once to encrypt a message. A single sender must use a new and unique nonce for each message. Multiple senders that want to use the same key must employ a scheme that divides the nonce space into non-overlapping sets, in order to ensure that the same nonce is *never* used twice. If two different messages are ever encrypted with the same (K,N) pair, Phelix loses most of its

security properties. We claim, however, that even in such a case it remains infeasible to recover the key.[4]

– The receiver may not release the plaintext P, or the keystream, until it has verified the tag successfully. In most situations, this requires the receiver to buffer the entire plaintext before it is released.

These requirements seem restrictive, but they are in fact required by all stream ciphers and many block cipher modes (e.g., OCB [RBBK01b,RBBK01a] and CCM [WHF]).

Although Phelix allows the use of short keys, we strongly recommend the use of keys of at least 128 bits, preferably 256 bits.

6 Other modes of use

So far we have described Phelix as providing both encryption and authentication. Phelix can be used in other modes as well. For any particular key, Phelix should be used with a single mode. Using several modes with a single key can lead to a loss of security.

6.1 Additional Authentication Data (AAD)

In packet environments, it is often desirable to authenticate the packet header without encrypting it. From the encryption/authentication layer, this looks like an additional string of data that is to be authenticated but not encrypted.

Let $\ell(A)$ be the number of such unencrypted header bytes to be authenticated. Phelix defines a simple extension for processing AAD that seamlessly reverts to the non-AAD case when $\ell(A) = 0$, defined as follows.

The AAD is padded with 0–3 zero bytes until the length is a multiple of four. After cipher initialization (section 3.4), the padded AAD blocks are "encrypted" as plaintext blocks, but all the associated ciphertext words are discarded. However, both before the first AAD block and after the final AAD block, the state variable Z_1 (i.e., $Z_1^{(0)}$ and $Z_1^{(\lceil \ell(A)/4 \rceil)}$) is XORed with the 32-bit hex constant 0xaadaadaa. Next, the input message data is encrypted (or decrypted) as a normal message through Phelix, except that the starting block number is $\lceil \ell(A)/4 \rceil$. At this point, after the last message word is processed (i.e., at the same time $Z_0^{(k)}$ is XORed with the value 0x912d94f1, as in section 3.6), the following steps are also performed, with $k = \lceil \ell(A)/4 \rceil + \lceil \ell(P)/4 \rceil$:

$$Z_2^{(k)} := Z_2^{(k)} \oplus \lfloor \ell(A)/2^{32} \rfloor$$
$$Z_4^{(k)} := Z_4^{(k)} \oplus (\ell(A) \bmod 2^{32})$$

After these updates to the internal state, the final MAC processing proceeds as in section 3.6, resulting in a MAC tag computed over the AAD and plaintext.

[4] Note that Helix actually was vulnerable to a key recovery attack if (Key,Nonce) pairs where reused [Mul04].

6.2 Truncated MAC Values

To reduce bandwidth consumption, it is often desirable to use a MAC value shorter than 128 bits. For example, in IPsec packets, MAC values (or "tags") are typically truncated to 96 bits. Phelix allows the use of truncated MAC tags, with some restrictions discussed below. If desired, a single Phelix key may be used with both MAC truncation and AAD without affecting security.

Let t be the desired size, in bits, of the MAC tag, with $0 < t \leq 128$. The definition of X'_i in section 3.3 is then extended, for the case $i \bmod 4 = 1$, to $X'_i := 4 \cdot \ell(U) + 256 \cdot (t \bmod 128)$. In other words, this technique embeds both $\ell(U)$ and t into different bit fields within X'_i. Note that the extended definition seamlessly reverts to the non-truncated case when $t = 128$.

After encryption, the truncated MAC tag consists of the first t bits of the 16-byte MAC value. More formally, let the untruncated MAC bytes be $(m_0, \ldots, m_{15})$. The truncated MAC bits are then the bytes

$$(m_0, \ldots, m_{\lfloor (t-1)/8 \rfloor - 1}\ ,\ m_{\lfloor (t-1)/8 \rfloor} \bmod 2^{1+((t-1) \bmod 8)}).$$

Given the nature of Phelix, where plaintext can affect the internal cipher state, MAC truncation must be handled prudently. As an extreme example in the absence of any restrictions, if a tag length $t = 1$ were negotiated, an attacker could submit forged packets with a repeated nonce value and have them successfully decrypted with probability 0.5, probably opening the door to devastating key recovery attacks. Fortunately, truncated Phelix MACs can be used safely, but only under certain restrictions:

- The receiver must implement an anti-replay strategy, so that multiple packets with the same nonce will never be decrypted. Since anti-replay is almost always a required policy anyway, this restriction is not onerous.
- The tag length t must be negotiated together with the key U, and the two values must always be bound together. The receiver must always compare all t bits of the MAC tag before "approving" a packet and revealing its plaintext. In other words, an attacker must not be allowed to submit packets to be encrypted or decrypted with a tag length t different from that negotiated when the key U was selected.
- Since an attacker will be able to mount an attack due to tag truncation only if he is able to have forgery attempts successfully decrypted, the tag length t must be large enough that there is no acceptable probability of having the receiver accept and decrypt a forged packet within the lifetime of the system.

This last restriction deserves some further discussion. For example, suppose that the receiver can verify and decrypt no more than 2^q packets per second, and that the desired security lifetime is 100 years, or approximately 2^{32} seconds. The probability of a forgery succeeding in the lifetime is then $p \approx 2^{q+32-t}$, so a value $t > q + 64$ is required to guarantee $p < 2^{-32}$. As a particular case of interest, on a 10-Gbit/sec link using IPsec with a minimum packet size of 64 bytes, we find $q \approx 21$, so we require $t > 85$ to achieve $p < 2^{-32}$. In practice, of course, it is almost certain that a successful attack

would require considerably more than a single forged packet, so larger values of p may well be acceptable. For example, a less demanding hypothetical system might use an 802.11 wireless link with $q < 2^{14}$. If the goal is only $p < 2^{-8}$ (so that the probability of getting more than four forged packets is easily less than 2^{-32}) and the expected system lifetime is 25 years, then the requirement could be met with $t > 54$.

Another common approach to enforcing such restrictions is to close a Phelix "session" (and negotiate a new key) after more than a certain number of forgery attempts have been detected by the decryptor. Alternately, a policy of rekeying after a certain number (e.g., 2^{32}) of packets have been processed will shorten the lifetime of a key and thus limit the probability of a forgery being accepted. The particular settings chosen for any such policies should be carefully chosen as part of the system design in order to meet the desired security goals without affecting performance.

As a general guideline, if truncated Phelix tags must be used, values of $t < 96$ should be considered only after careful analysis, and values of $t < 64$ are strongly discouraged in all cases.

6.3 Pure stream cipher, PRNG, and PRFG

Phelix can be used as a pure stream cipher by ignoring the MAC computations at the end. And like any stream cipher, Phelix is a cryptographically strong pseudorandom number generator. For every (key,nonce) input, it produces a stream of pseudorandom data. This makes Phelix suitable for use as a PRNG. Similarly, Phelix can be viewed as a Pseudorandom Function Generator: given a secret key and encrypting an all-zero plaintext, the nonces define different outputs, indistinguishable from uniformly chosen independent bit strings.

6.4 MAC with Nonce

Phelix can also be used a pure MAC function. The data to be authenticated is encrypted, but the ciphertext is discarded. The receiver similarly discards the keystream and just feeds the plaintext to the Phelix rounds. In this mode Phelix is significantly faster than, for example, HMAC-SHA1, but it does require a unique nonce for each message. Unfortunately, it is insecure to use Phelix with a fixed nonce value.

7 Design rationale

Although the design strength of Phelix is 128 bits, we use 256-bit keys. This avoids a very general class of attacks that exploits collisions on the key value. For flexibility, Phelix also allows shorter keys to be used, as there are many practical situations in which fewer than 256 bits of key material are available.

The small set of elementary operations that Phelix uses makes it efficient on a large number of software platforms. The absence of tables, variable rotations, and multiplications makes Phelix small and efficient in hardware as well.

The number of active state words (i.e., 5) was chosen as the maximum state that can easily fit in the registers of mainstream Intel CPUs. The number of

old state words (4) used for output "masking" was chosen so that the total amount of unknown internal state is always at least 256 bits, even after the keystream output.

Most ciphers use lookup tables to provide the necessary nonlinearity. In Phelix, the nonlinearity comes from the mixing of XORs with additions. Neither of these operations can be approximated well within the group of the other.

The diffusion in Phelix is not terribly fast, but it is unstoppable. As the attacker has very little control over the state, it is not possible to limit the diffusion of differences. In those areas where dynamic attacks are possible, we use a sequence of 8 blocks to ensure thorough mixing of the state words.

The key mixing is an unkeyed bijective function. The purpose is to spread the available entropy over all key words. If, for example, the key is provided by a SHA-1 computation, then only 5 words of key material are provided. The key mixing ensures that all 8 key words depend on the key material. Using a bijective mixing function ensures that no two 256-bit input keys lead to the same working key values. The use of the input key length in X' guarantees that even keys that lead to the same working key (each short key leads to a working key that is also produced by a 256-bit key) do not lead to equivalent Phelix encryptions.

7.1 Key schedule

The $X_{i,0}$ values simply cycle through the key words. The $X_{i,1}$ values depend on the same key words in anti-phase, the extended nonce words, the block number, and the input key length. This key schedule has a number of properties. All 8 key words and all 4 nonce words affect the state every 4 blocks.

The key schedule also ensures that different (K, N) pairs produce different block key sequences. Even stronger: no sequence of 17 key words ever occurs twice across all keys, all nonce values, and all positions in the encryption computation.

To demonstrate this, we look at the sequence $W_j := X_{\lfloor j/2 \rfloor, j \bmod 2}$. This is the sequence of key words in the order they are used. Given just part of the sequence W_j, without the proper index values j, we can recover the key, nonce, and block number. (When the plaintext word is zero, the first half of the block function is identical to the second half of the block function, so it makes sense to look at the sequence W_j and allow half-block offsets.)

If $W_j = W_{j+16}$, then j is even; otherwise, j is odd. This allows us to split the Y values back into an $X_{i,0}$ and $X_{i,1}$ sequence.

Now consider

$$\begin{aligned} D_i &:= X_{i,1} - X_{i,0} + X_{i+4,1} - X_{i+4,0} \\ &= N_{i \bmod 8} + N_{(i+4) \bmod 8} + X'_i + X'_{i+4} + 2i + 20 \\ &= (i \bmod 4) + 2i + 20 + X'_i + X'_{i+4} \end{aligned}$$

all modulo 2^{32}. We first look at $D_i \bmod 4$. The X' terms can only have a nonzero contribution if $i \bmod 4 = 3$, so 3 out of 4 consecutive times we get just $((i \bmod 4) + 2i) \bmod 4 = 3i \bmod 4$, which gives us $i \bmod 4$. Looking at the full D_i for an i with $i \bmod 4 = 0$ gives us $i \bmod 2^{31}$. The sum $X'_i + X'_{i+4}$

from the case $i \bmod 4 = 3$ gives us the upper bits of i. This recovers[5] the block number, i.

Given $i \bmod 8$, we can recover the working key from the $X_{i,0}$'s. Knowledge of i, and the key words allows us to compute the key length and the nonce from the $X_{i,1}$'s, as well as check the redundancy introduced by the nonce expansion to 8 words.

We have not investigated whether it is possible to recover the key, nonce, and block number from fewer than 17 consecutive key words. A simple counting argument shows that at least 14 are required. This remains an open problem.

7.2 Choice of Rotation Counts

Phelix mixes additions mod 2^{32} and bit-wise XORs. The rotations provide the diffusion between the various bit positions in the state words. Without the rotations, the diffusion could only go into one direction: from the less significant bit positions to the more significant ones. Thus, the choice of rotation counts is critical for the security of Phelix. During the design process we examined the impact of various choices of rotation counts both in terms of attempts to cryptanalyze the cipher, and also in terms of their impact on statistical tests of the block function.

To analyze the diffusion properties of a set of rotation counts, consider a variant of the block function with all the additions changed to XORs. (This is equivalent to ignoring the carries in the additions.) In this variant, we can track which output bits are affected by which input bits. For this analysis, we consider an output bit affected if its computational path has a dependency on the input bit at any one point, even if the output bit in our linearized block function is not changed due to several dependencies canceling out. This seems to be the most suitable way to analyze diffusion and is related to the independence assumption in differential and linear cryptanalysis.

A set of rotation counts can, at best, ensure that changing a single state input bit affects at least 21 bits of the output. There are a large number (over 6 000) of such rotation count sets.

We discarded all rotation count sets that contained a rotation count of 0, 1, 8, 16, 24, or 31. Rotation by a multiple of 8 repeats after only two or four operations, and rotation by 1 or 31 bit positions provides diffusion between adjacent bits, something the carry bits already do. This reduced the set of candidate rotation counts to 86.

Using the full block function, we ran statistical tests on many candidate rotation count sets to see how these values would affect the ability of the block function to diffuse changes and mix together separate information within the 160-bit internal state. Among our tests, we considered:

1. The number of rounds required before all output bits passed binomial tests, given a fixed input difference in the state.

[5] This isn't absolutely perfect. We do not recover the 62'nd bit of $i + 8$, but this bit will only be set during the very last few blocks of a message very close to 2^{64} bytes long. This does not lead to a weakness.

2. The number of rounds required before the output states' Hamming weight distribution passed a χ^2 test, given low- and high-Hamming weight input states.
3. The number of rounds required before the output states' differences in Hamming weight distribution passed a χ^2 test, given low- and high-Hamming weight differences in the input state [KRRR98].
4. Low- and high-Hamming weight higher-order differences, and the number of rounds required before the resulting output differences' Hamming weights passed a χ^2 test.

The surprising result was that most rotation counts did pretty well. Our carefully selected rotation count sets were slightly better than random ones, but only by a small margin. Degenerate rotation counts (all rotation counts equal, or most rotation counts zero) led to much worse test results.

At the end of our analysis, we selected more or less at random from the remaining candidates. Based on our limited analysis, the specific choice of rotation counts does not have a strong impact on the security of Phelix, with only the caveat that we had to avoid some obvious degenerate cases.

8 Cryptanalysis

Phelix is intended to provide everything needed for an encrypted and authenticated communications session. A successful attack on Phelix will have occurred when an attacker can either predict a keystream bit he hasn't seen with a probability slightly higher than 50%, or when he can create a forged or altered message that is accepted by the recipient with a probability substantially higher than 2^{-128}. To be meaningful, given the 128-bit security bound of Phelix, any such attack must require fewer than 2^{128} block function evaluations for all participants combined. Also, any such attack must obey the security requirements placed on Phelix operations; e.g., no reuse of nonces, MACs checked before decrypted messages released, etc.

In this section, we consider a number of possible ways to attack Phelix. Although our time and resources have been limited, we have not yet discovered any workable method of attacking Phelix.

8.1 Static analysis

A static analysis takes the keystream and tries to reconstruct the state and key. Several properties make this type of attack difficult. Even if the whole state is known, any four consecutive keystream words are fully random. This is because each $X_{i,0}$ key value affects S_i in a bijective manner, so for any given state and any sequence of $X_{i,1}$ words, there is a bijective mapping from $K_{i \bmod 8}, \ldots, K_{(i+3) \bmod 8}$ to $S_i, \ldots, S_{i+3}$. A similar argument applies when the block function is computed backwards. Any attempt to recover the key, even if the state is known at a single point, must therefore span at least 4 blocks. Of course, there is no reasonable way of finding the state. At any point in a block, there are at least 256 bits of unknown state, even after the keystream word is output. As the design strength is 128 bits, an attacker cannot afford

to guess the entire state. A partially guessed state does not help much, as key material is added at twice the rate that keystream is produced.

8.2 Period length

The Phelix internal state is updated continuously by the plaintext it is encrypting. So long as the plaintext is not repeating, the keystream should have an arbitrarily long period, up to the maximum packet size of 2^{64} bytes.

With a fixed or repeating plaintext, the Phelix state does not cycle, either. In section 7.1 we showed that any 17 consecutive key words used as inputs to the block function are unique. The nonrepeating key word values prevent the state from ever falling into a cycle.

8.3 State collisions

In [Mul04], an internal collision after 2^{114} words of chosen plaintext was found in the cipher Helix [Helix03], the predecessor to Phelix. To avoid this class of attack, Phelix added the 4 "old" state words, increasing the internal state significantly to 288 bits. Thus, the unknown state remains at 256 bits even after outputting the keystream word within a block, so no collisions are reasonably expected within the security bounds.

8.4 Weak keys

Phelix makes constant use of the words of the working key. An all-zero working key intuitively seems like a bad thing (it effectively omits a few operations from the block function), but we have not discovered any possible attack based on it. The all-zero working key is only generated by a single key of 32 bytes in length. Shorter key length cannot generate the all-zero working key. The all-zero working key does not seem to have any practical security relevance, and there is no reason to treat this key differently from any other key.

8.5 Adaptive chosen plaintext attacks

Because the plaintext affects the state, Phelix allows an attack model that traditional stream ciphers prevent: An attacker can request the encryption of a plaintext block under an already established (key,nonce) pair, and can use the resulting ciphertext to determine what plaintext to request next.

We have found no way to use such an attack against Phelix. As with the discussion of static analysis above, the large unknown and untouchable state, and the continual mixing of key material into that state, appear to defeat attempts to use control over one input of the block function to control other parts of its state. Additionally, the usage restrictions on Phelix do not allow reuse of nonces, which ensures that the state is always a "moving target."

However, it should be noted that in [Mul04], a key recovery attack of complexity 2^{88} was found against Helix [Helix03], assuming nonce reuse. While this attack clearly violates critical and commonly accepted usage restrictions of stream ciphers, the concern that the Helix key itself could be recovered led

to the Phelix enhancements. The active state values of Phelix are identical to those of Helix, but the inclusion of the "old" state variables in the Phelix output keystream function makes it much more difficult to guess or learn the internal active state than in Helix. Further, the point within the Phelix block at which the keystream is extracted, namely $Y_4^{(i)}$, forces much greater diffusion and nonlinear mixing of the previous plaintext word, thwarting the differential plaintext attack described in [Mul04].

8.6 Chosen input differential attacks

One powerful mode of attack is for the attacker to make small changes in the input values and look at how the changes propagate through the cipher.

In Phelix, this can be done only with the key or the nonce. In each case, the block function is applied multiple times to the input. In Phelix, all the places where such attacks are possible have eight consecutive blocks without any output. A change to the nonce, such as is considered in [DGV93], will be thoroughly mixed into the state by the time the first keystream word is generated. Similarly, a change to the last plaintext byte is thoroughly mixed into the state before the first MAC tag word is generated. A differential attack would have to use a differential through 8 blocks, or 160 rounds of Phelix. A search found no useful differentials for 8 blocks of Phelix, nor useful higher-order differentials.

8.7 Algebraic attacks over GF(2)

The algebraic analysis of Helix, the predecessor to Phelix, employed linearization techniques, inspired by extended linearisation [CP02]. Independently from Helix, such algebraic techniques have been used to successfully analyze stream ciphers [Cou02,Arm02]. The best attack against Helix was based on optimistically (from the addversary's point of view) assuming a certain system of equations to have full rank. The adversary than had to solve this system with its $\approx 2^{49.1}$ binary unknowns. Solving a system of linear equations in N unknowns by Gaussian elimination takes time $O(N^3)$. However, [CP02] suggested a $O(N^{2.376})$-time algorithm, though with some apparently huge proportionality constant. If we extremely optimistically set this constant to 1, the attacks takes time $2^{116.7}$. At the time of designing Helix, we did not consider this a practical threat, and, due to the optimistic and even unrealistic nature of our assumptions, we conjectured the algebraic attack actually to to take more than 2^{128} steps. We are not aware of any results contradicting this conjecture.

As it turns out, Phelix' resistance against this type of attack greatly improves on Helix' resistance. While Phelix uses the same block function as Helix, the internal state size of Phelix has increased by 128 bits. The analysis shows that (under the same full-rank assumption as before), the system of linar equations now has $\approx 2^{64.07}$ unknowns. No algorithm is known, which could possibly solve such a system of linear equations in 2^{128} steps or less.

9 Conclusions and intellectual property statement

Most applications that require symmetric cryptography actually require both encryption and authentication. We believe that the most efficient way to achieve this combined goal is to design cryptographic primitives specifically for the task. Towards this end, we present a new such cryptographic primitive, called Phelix. We hope that Phelix and this paper will spur additional research in authenticated encryption stream ciphers. As with any experimental design, we note that Phelix should not be used until it has received additional cryptanalysis.

There are no hidden weaknesses inserted the the designers of Phelix, nor are we aware of any material weaknesses of the algorithm.

Finally, we hereby explicitly release any intellectual property rights to Phelix into the public domain. Furthermore, we are not aware of any patent or patent application anywhere in the world that covers Phelix.

10 Acknowledgements

We would like to thank David Wagner, Rich Schroeppel, Niels Ferguson, John Kelsey, and Yoshi Kohno for their helpful comments and encouragements during the development of Helix and Phelix.

References

[Arm02] Frederik Armknecht. A linearization attack on the Bluetooth key stream generator. Cryptology ePrint Archive, Report 2002/191, 2002. `http://eprint.iacr.org/2002/191`.

[Cou02] Nicolas Courtois. Higher order correlation attacks, XL algorithm, and cryptanalysis of Toyocrypt. In *Information Security and Cryptology–Icisc 2002*, volume 2587 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. To appear.

[CP02] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Yuliang Zheng, editor, *Advances in Cryptology—ASIACRYPT2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer-Verlag, 2002.

[DGV93] Joan Daemen, René Govaerts, and Joos Vandewalle. Resynchronisation weaknesses in synchronous stream ciphers. In Tor Helleseth, editor, *Advances in Cryptology—EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 159–167. Springer-Verlag, 1993.

[Gol00] Jovan Dj. Golić. Modes of operation of stream ciphers. In Douglas R. Stinson and Stafford Tavares, editors, *Selected Areas in Cryptography, 7th Annual International Workshop, SAC 2000*, volume 2012 of *Lecture Notes in Computer Science*, pages 233–247. Springer-Verlag, 2000.

[Helix03] Niels Ferguson, Doug Whiting, Bruce Schneier, John Kelsey, Stefan Lucks, and Tadayoshi Kohno. Helix: Fast encryption and authentication in a single cryptographic primitive. In Thomas Johansson, editor, *Fast Software Encryption, 10th International Workshop, FSE'03*, volume 2887 of *Lecture Notes in Computer Science*, pages 330–346. Springer-Verlag, 2003.

[Jut01] Charanjit S. Jutla. Encryption modes with almost free message integrity. In Birgit Pfitzmann, editor, *Advances in Cryptology—EUROCRYPT2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 529–544, 2001.

[KRRR98] Lars R. Knudsen, Vincent Rijmen, Ronald L. Rivest, and M.J.B. Robshaw. On the design and security of RC2. In Serge Vaudenay, editor, *Fast Software Encryption, 5th International Workshop, FSE'98*, volume 1372 of *Lecture Notes in Computer Science*, pages 206–221. Springer-Verlag, 1998.

[LM01] Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. In Mitsuru Matsui, editor, *Fast Software Encryption2001*, Lecture Notes in Computer Science. Springer-Verlag, To appear, 2001. Available from `http://www.tcs.hut.fi/~helger/papers/lm01/`.

[Mul04] Frédéric Muller. Differential Attacks against the Helix Stream Cipher. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE'04*, volume 3017 of *Lecture Notes in Computer Science*, pages 94–108. Springer-Verlag, 2004.

[RBBK01a] Philip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption, September 2001. Available from `http://www.cs.ucdavis.edu/~rogaway`.

[RBBK01b] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *Eighth ACM Conference on Computer and Communications Security (CCS-8)*, pages 196–205. ACM Press, 2001.

[WHF] Doug Whiting, Russ Housley, and Niels Ferguson. Counter with CBC-MAC (CCM). Available from `csrc.nist.gov/encryption/modes/proposedmodes/ccm/ccm.pdf`.

A Test vectors

The authors maintain a website at `http://www.schneier.com/phelix.html` with Phelix news, example code, and test vectors. We give some simple test vectors here. (The 8-word working key is given as a sequence of 32 bytes, least significant byte first.)

```
MAC tag:     128 bits
Initial Key: <empty string>
Working Key: A9 3B 6E 32  BC 23 4F 6C  32 6C 0F 82  74 FF A2 41
             E3 DA 57 7D  EF 7C 1B 64  AF 78 7C 38  DC EF E3 DE
Nonce:       00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
AAD:         <empty string>
Plaintext:   00 00 00 00  00 00 00 00  00 00
Ciphertext:  D5 2D 45 C6  05 FD 7A 67  74 8D

MAC tag:     128 bits
Initial Key: 00 00 00 00  01 00 00 00  02 00 00 00  03 00 00 00
             04 00 00 00  05 00 00 00  06 00 00 00  07 00 00 00
Working Key: 6E E9 A7 6C  BD 0B F6 20  A6 D9 B7 59  49 D3 39 95
             04 F8 4A D6  83 12 F9 06  ED D1 A6 98  9E C8 9D 45
Nonce:       00 00 00 01  01 00 00 01  02 00 00 01  03 00 00 01
AAD:         <empty string>
Plaintext:   00 01 02 03  01 02 03 04  02 03 04 05  03 04 05 06
             04 05 06 07  05 06 07 08  06 07 08 09  07 08 09 0A
Ciphertext:  B5 FC 4B F5  BC 64 0A 56  00 3D 59 6D  33 4B A5 94
             A5 48 7B 4E  30 8E DB 05  A7 D6 2F 23  45 14 02 4A

MAC tag:     64  bits
```

```
Initial Key: 01 02 03 04  05 06 07 08  08 07 06 05  04 03 02 01
Working Key: 98 3F 23 CE  B9 F4 D9 68  B0 56 54 06  2D 15 34 C6
             4B 38 AD E7  C2 BA A4 DF  D8 D4 02 21  DB BC 96 E8
Nonce:       04 00 00 00  05 00 00 00  06 00 00 00  07 00 00 00
AAD:         <empty string>
Plaintext:   <empty string>
Ciphertext:  <empty string>

MAC tag:     96  bits
Initial Key: 09 07 05 03  01
Working Key: 36 0A 5B CD  91 EC 72 89  0A C3 BA 9D  1C 48 E2 E0
             03 1C 86 33  83 A4 9F D8  CB F8 CC CA  1F D6 AB 5D
Nonce:       08 07 06 05  04 03 02 01  00 01 02 03  04 05 06 07
AAD:         00 02 04 06  01 03 05 07  08
Plaintext:   00 01 02 03  01 02 03 04  02 03 04 05  FF
Ciphertext:  F1 0D 3E 06  7A 32 B1 BE  DA A5 89 8B  DE
```