---------------------- hpc-overview ----------------------

An Overview of the Hasty Pudding Cipher

Rich Schroeppel & Hilarie Orman

rcs@cs.arizona.edu

July 1998

While we are all waiting for NIST to uncloak the new Advanced

Encryption Standard, we've cooked up a tasty morsel: the Hasty Pudding

Cipher (HPC) is fast and flexible, and provides strong security. A

bottle of Dom Perignon to anyone who cracks the cipher!

This is a public domain method with no known patents. The code is not

copyrighted.

Hasty Pudding is a variable length block cipher. The block size may

be \*any\* number of bits, even fractional bit values are permitted.

The arbitrary block size means that anything can be encrypted without

expansion. It also means that large data units (files, email

messages, etc.), if encrypted as single blocks, can have encryptions

that interrelate all of the constituent bits, not merely those that

are in contiguous 64 bit blocks, for example.

The key size may be any whole number of bits; the key space is larger

than any measure of the physical universe. It is possible to use a

long-term key, with key lifetime not limited by the "amount of

material protected by one key" restriction. Of course, other lifetime

limitations will still apply -- keys can be stolen, or leak, etc.

Hasty Pudding is fast, achieving 100 megabits per second on 1000-bit

blocks on a 233 MHz DEC Alpha. The cipher works best on 64-bit

architectures, but runs acceptably fast on 32 bit machines. For RISC

architecture machines, such as the Alpha, the cipher is remarkably

conservative of instruction usage: on the Alpha it uses about 20

instructions and 2-3 memory references per byte of input plaintext.

The Hasty Pudding Cipher has a security feature applied to each

datablock, the \*spice\*. It may be regarded as a secondary key that

need not be concealed. Use of the spice offers one of the

advantages of cipher block chaining mode (CBC), because the spice can

be changed very cheaply for each block encrypted, yielding different

ciphertext even when the plaintext remains constant. Spice offers an

advantage of over CBC, though, because the spice can be generated

quickly and used for parallel encryption of several blocks at once.

The cipher is fast enough to be used for many distributed system

application, such as video multicast, virtual memory for diskless

client machines, and encrypted file systems. Interestingly, random

access is well supported, because each record (or file block) can be

encrypted with a different spice, perhaps based on block number or

customer account number. For structureless text files, the block

number could be used as the spice. Because there is no expansion

(ciphertext size is always identical to plaintext size), and no CBC

problem, a new block can be written over an old one without the

block-to-block chaining problem that CBC has.

The architectural dependencies of the cipher speed are minimal. A

RISC machine with 64 bit registers and 64 bit memory operations, a

first-level data cache of at least 10K bytes, and an instruction cache

of at least 8K bytes will have the best performance, but similar 32

bit architectures will have acceptable performance as well. The

cipher is quite miserly in its occasional use of multiplication and

other expensive instructions. Pipeline busting conditional branches

are rare.

The disadvantages of the cipher include its complexity and resulting

use of a large amount of instruction memory. Another disadvantage for

power-limited processing environments is its use of RAM: for each key

it requires 2K per blocksize range, and there are 5 blocksize ranges.

Performance Considerations

The highest speeds are for blocks of length 64 bytes and up. The

shorter blocksizes are slightly slower. The speed for single bits is

much slower but still compares favorably with other methods. In

actual practice, cipher speed is important for bulk encryption, but

less important for small data units, because the cost of processing

small blocks is generally dominated by application service

requirements (parsing, computation, disk access, etc.). Thus, HPC is

engineered to be fastest where speed counts most.

The extra effort to safely include the spice impacts noticeably on the

time for short blocksizes; even if the spice is removed, the cipher

remains sluggish. A user merely needing to encrypt single bits might

instead choose to encrypt 64 bits of zero at a time, thereby obtaining

64 bits to XOR with the target.

Because the cipher accesses the datablocks at random, very large

datablocks (e.g. one million 64 bit words) may cause translation

lookaside buffer misses that cause the performance to degrade

slightly.

The code size might exceed the size of the on-chip cache for some

architectures; this will have an adverse effect on performance.

When implemented on a 32-bit computer architecture, the speed will be

greatly affected by the operations that add, shift, and rotate,

because the testing for carry bits will cause the instruction pipeline

to stall.

The running time of the algorithm depends on several usage factors,

one of them being the length of the spice. If the spice is shorter,

the algorithm will run faster. This offers opportunities for

fine-tuning the speed of the algorithm to the security requirements of

the application.

When the cipher is used for encrypting data values that have a domain

size that is not a power of two, the running time of the algorithm is

probabilistic. This is due to the method that Hasty Pudding uses for

mapping from the domain to a cipher range that is nearly equal in

size: encipherments are generated in a larger domain until a value

within the domain appears. This assures that the cipher values have a

uniform range, the same as the plaintext domain values. However, the

variance in the running time may not be acceptable for hard deadline

applications.

The five key expansion tables needed for each key imply that a data

cache of at least 10K bytes is normally required. For specialized

applications, utilizing only one block size, only a single 2K table

need be generated; this reduces instruction space usage as well. The

fractional bit techniques might also be dropped in some cases, further

reducing the instruction memory requirements.

Design Considerations

Though this is a "kitchen sink" cipher, using many disparate tricks,

the irregularity has its advantages. It prevents differential

cryptanalysis and makes linear analysis hard. The analysis is harder

for both the good guys and the bad guys. Even in the event of

practical quantum computing methods emerging in the future, Hasty

Pudding is likely to fare well because of its ad hoc design

complexity; it may well have too much state to represent in any

realizable quantum machine. On the other hand, this complexity means

that the cipher cannot be represented compactly for today's computer

architectures, either, and the cipher will have a large instruction

footprint.

A design goal for the cipher is to have triple diffusion: one bit of

change in data or spice propagates three times through the cipher

state, changing every bit. Some of the subciphers have diffusion

near five. The extended-length subcipher has diffusion two, but

is compensated by the mixing rounds.

The Cipher uses the non-linearity of addition and exclusive OR to

achieve mixing of the bits. The key expansion memory serves as an

additional non-linear function, mapping 8 bits to 64 bits.

The key expansion table is referenced at least 24 times during any

block encryption, utilizing 64 bits from the table on each access.

This ensures a liberal use of keying material in each block of output

ciphertext.

A single bit change in spice affects all ciphertext bits with nearly

equal probability. In the event that Hasty Pudding is used with a

requirement for secrecy of the spice, this feature minimizes the

likelihood that cryptanalysis of the ciphertext could reveal the spice

values.

The cipher is designed to take advantage of instruction-level

parallelism, and the spice allows block level parallelism.

Overview of the Operation of the Hasty Pudding Block Cipher

The inputs to the main HPC routine are the following:

Plaintext

Output buffer (this is the output area)

Length of the plaintext (in bits)

Key

Length of the key (in bits)

Array of 5 pointers to key expansion tables

Pointer to spice

Multiple encryption count

Hasty Pudding is five different ciphers for five different block size

ranges:

tiny <=35 bits

short 36-64 bits

medium 65-128 bits

long 129-512 bits

extended >=513 bits

Each cipher has the underlying Feistel structure (although it's

changed so much that Feistel probably wouldn't recognize it).

The cipher key controls the key expansion table at the start of the

cipher.

The internal state of the cipher contains up to 8 variables of 64 bits

(512 bits), depending on the block size. The internal state of the

cipher is initially derived from the plaintext.

The cipher consists of a series of steps (a "step" is similar to a DES

"round") that alter the values of the internal state variables. A

step alters each state variable at least once. Each of the

assignments to a state variable is called a microstep, and each

microstep is a simple, reversible function of its inputs.

Many steps use eight bits of the state to select a word from the key

expansion table. The word is mixed with some the state via the

microsteps, and the state is then "stirred" by mixing it with itself.

Every few steps, the Cipher uses the spice to alter the state. This

pattern (cipher, spice) is run for a few steps; the spice is used a

minimum of five times for each block.

A microstep takes as inputs the state variable that will be altered,

and one or two other selected words from the state, and possibly one

or two words from the key expansion table. The operations for

combining the inputs are selected from the following set: addition,

subtraction, exclusive or. Often one of the inputs is shifted.

Decryption consists of reversing the microsteps.

The extended cipher works by interleaving a step with an operation

that exchanges a plaintext word with a state word. This combination

of step and exchange is executed once for each word in the plaintext

array. This is called a pass, and there are three passes in extended

mode. Each pass uses the plaintext array in a different order.

Between passes, a few steps are executed to make sure that small

changes in the plaintext propagate throughout the ciphertext. Some

steps use the spice. Each step is invertible.

The inputs to the HPC function for fractional bit encryption are the

following:

Plaintext (this is the integer that is to be encrypted)

Size of the output range (an integer)

Key

Length of the key (in bits)

Array of 5 pointers to key expansion tables

Pointer to spice

Multiple encryption count

The output of the fractional bit encryption is an integer between zero

and the size of the output range minus 1.

Key Expansion

There are five key expansion tables, one for each subcipher. Each

table is an array of 256 64-bit words. The code for creating the

tables is the same for each subcipher. The table initialization

depends on the subcipher number, which produces completely different

expansion arrays.

There are three inputs to the key expansion process: the key, the

subcipher number (one to five), and the low 64 bits of the length of

the key. The inputs control the initialization of the array. The

stirring process amplifies minor differences. In the version of Hasty

Pudding coded below, the key expansions are done only when needed.

An application may use one key expansion table for all five

ciphers. In this case, the table for medium length is used. This

carries some additional risk, since it allows the possibility that one

of the five ciphers could be broken, and the key expansion table

recovered, which would then expose traffic in the other four ciphers.

Modes

Because it is a block cipher, Hasty Pudding can be used in all the

usual block cipher modes.

The spice itself offers all the advantages of CBC mode as well some

unique advantages, such as parallel encryption.

Hasty Pudding can be used to simulate a stream cipher by initializing

the spice to zero, and then incrementing it for each item encrypted.

The efficiency is of this mode is poor for bits, but is tolerable for

bytes, and reasonable for 64-bit words.

If a "random" bit stream is all that is required, then encryption of

any convenient size of integers will do.

Implementations of Hasty Pudding must include the option for multiple

encryption modes. In the event that machine speeds or cryptanalysis

renders the single encryption mode less secure than is desirable, it

should be possible to easily change fielded versions to use double or

triple encryption mode.

The Spice

Each block can be encrypted with a unique contribution, the spice.

This is similar to an initialization vector (IV) for DES, in that it

need not be concealed and it assures that two identical plaintext

blocks will not have the same ciphertext. The spice need not be used,

but if it is, it protects against block splicing attacks that are

common to the cipher block chaining (CBC) modes.

The spice also allows parallel encryption of several blocks, a

capability that CBC prevents. Both CBC and Hasty Pudding with spice

can be parallel decrypted.

The spice is long enough (512 bits) that it can be used to contain a

variety of "uniquifying" data. These might include things like date

and time, inode number, user account number, filename, etc. One word

can be reserved for the block number, so any file or data connection

will have unique encryption of every block sent. Even if the

plaintext is a constant stream of zeroes, a different spice for each

block will cause the ciphertext to look random.

Block splicing is not a problem, because moving a ciphertext block to

another place will cause it to decrypt to randomness. This is better

than CBC, which can be spliced at the cost of two random decrypted

blocks.

The spice can be shortened to fewer words, or deleted entirely, for

extra speed. Unused portions of the spice are treated as 0.

Spice Caveat

Avoid using spice that is controlled by an opponent because the cipher

is not warranteed against a chosen-spice attack. Though no such

attack is known at this time, it seems prudent to avoid this mode

absent more analysis. The spice is not a substitute for a cipher key,

and the cipher should never be used in a mode where the key is known.

Security Analysis

I'm claiming a security level of 400 bits. This means that an

attack will require 2^400 trials to succeed. The claim is based on

the amount of intermediate state in the cipher, and the amount

of key expansion table used for each encryption.

Five separate key expansion (KX) tables are used. Each is 256 64-bit

words. One security goal is that a break in one algorithm which

reveals the KX table doesn't spoil the other sub-algorithms. The KX

algorithm has been made deliberately lossy, so that an attacker who

learns a KX table cannot work backward to find the original key.

For single bits, knowing the encryption of zero implies the encryption

of one and vice-versa. For somewhat larger blocks of size B, if the

attacker learns the encryptions of 2^B-1 values, then the last value

is determined, as is the parity of the defined permutation.

The step used in Hasty Pudding churns most of the bits. A one-bit

change will be amplified to 2 or more bits changed. After 9 steps, a

one-bit change can affect all 2^9 state bits. These include the KX

lookup, and the variable shifts.

The cipher includes both linear and non-linear combining operations:

xor and add/subtract. Xor is bitwise linear, but arithmetically

non-linear. Viewed from an arithmetic perspective, add/subtract is

linear, but xor is non-linear. (In both cases, shifts are nearly

linear.)

The KX lookup is highly non-linear.

Two other mixes are relevant: Any single bit position within a word

(or the entire state) is mostly mixed with other words in the same

position. The variable shifts, and the fixed shifts, guarantee

diffusion along the bit-position dimension.

Finally, adjacent collections of bits must be broken up. This is done

by allowing parts of words to fall of the ends during shifts.

State information flows from each word into each other word; three

steps gives a complete graph of words flowing into every other word

and themselves.

The spice is mixed in at a time when the state from the initial

plaintext (or final ciphertext) is so thoroughly mixed that nothing

can be learned. If the attacker can somehow peer into the internal

state (or guess it), and vary the spice to cancel it, the other spice

uses will scramble the effort, foiling attacker's attempts to detect

the match.

If the attacker can guess most of the KX array, then a sufficient

number of known plaintext-ciphertext pairs will fill in the array.

Varying the spice will somewhat blunt this attack.

If 16384 bits of plaintext-ciphertext are known, along with the spice

values used, then the KX array is in theory determined. If the spice

is fixed but unknown, 16896 bits (in theory) define both the KX and

the spice.

Although indefinite size keys are allowed, there are in effect only

2^16384 distinct keys. (If all five tables are considered, then there

are 81920 bits of key.)

It is important that the key expansion table be generated by the

pseudorandom process in the cipher specification and not by direct

user input. This is because a small change in the expanded key

might not have any effect on the ciphertext. In such a case, an

attacker might guess which key words were used, and vary the spice to

try to solve for them.

Challenge

In keeping with the tradition of offering prizes for cryptographic

success, while reflecting my modest means, I (RCS) am offering a bottle

of Dom Perignon champagne for progress attacking Hasty Pudding. I will

attempt to award a prize each year for the best work that comes to my

attention. You don't have to break Hasty Pudding to win -- the best

paper may be analytical, or statistical, or attack a simplified

cipher, or the key expansion, or suggest an improvement to the cipher.

The prize work must be publicly available. I'll make awards until the

cipher is broken or I have given ten prizes.

---------------------- hpc-nist-doc ----------------------

The Hasty Pudding Cipher: Specific NIST Requirements

Rich Schroeppel

rcs@cs.arizona.edu

June 1998

Blocksizes and Key Sizes Supported

The Hasty Pudding Cipher supports all key sizes, from 0 bits on up.

An entire file may be used as a key if desired. Changing any bit

of the key produces completely different encryptions. Changing the

key length by appending 0's produces completely different encryptions.

The cipher supports a secondary key called the spice. The spice may

be up to 512 bits long. Short spices are 0-padded to full length.

The spice may be changed in one instruction. Changing any bit of

the spice produces completely different encryptions.

The spice need not be concealed from an opponent.

The key and the spice are independent: Key-expansion does not depend

on the spice.

The cipher supports all blocksizes, from 0 bits on up. Encryption

does not expand or pad the data. The cipher is efficient on megabyte

blocks. (The cipher is less efficient if the block doesn't fit in

memory.) Changing any bit of the plaintext produces a completely

unrelated ciphertext -- even changing the last bit of a megabyte block.

The cipher also supports blocksizes that are not an integral number

of bits. Any size numerical range may be encrypted. For example,

dates may be encrypted into dates. This is also a ``no-expansion''

operation.

The cipher can even be used to encrypt an arbitrary set to itself, such

as the set of printable ASCII characters, or the set of 86-bit primes.

All combinations of key size, spice size, and blocksize are supported.

Expected Strength

I have designed the Hasty Pudding Cipher to have a raw strength of 400 bits.

Except for the (obvious) situations outlined below, an attacker will need

computational effort 2^400 to

(a) from any number of given plaintext ciphertext pairs,

recover an expanded key table, or an original key, or a spice value

(b) from any set of expanded key tables, to calculate a different expanded

key table, or recover an original key

(c) from any number of plaintext ciphertext pairs, determine the

encryption or decryption of a different datum,

(c2) or estimate the probability that a hypothetical P-C pair is correct

(d) determine the length of a key

(e) determine any bit of the key

even if the attacker knows the spice value.

If the spice is partly known, the attacker will be unable to determine

other bits, except by exhaustive search over all the unknown bits.

The claim also applies when the cipher is used with non-integral

blocksizes (see caveat below).

Exceptions:

If a B bit block is being encrypted, and the attacker knows 2^B-1 P-C

pairs, the remaining P-C pair is determined. (This is usually only

relevant when B=1, but might apply for other small B.)

If the attacker knows or guesses the key length, he can try all possible

keys. For the NIST preferred key sizes of 128, 192, and 256 bits, the

attacker will be able to break the cipher in an average of 2^127, 2^191,

or 2^255 trials.

If the attacker learns the entire P-C map, he can determine whether an

odd or even permutation is specified, which will allow determination of

one parity bit of an intermediate collection of internal cipher state bits.

I have also designed the cipher to withstand a chosen-spice attack,

but I am uneasy about this possibility. I do not recommend the

algorithm be used in ways that would allow the opponent to control the

spice value. (Since the spice is cheap to change, it is reasonable to

use a different value for every encryption, completely trumping this

concern.)

If the attacker knows the key but not the spice, it will be hard

to determine the spice, but I haven't determined how hard.

Caveat for non-integral blocksizes:

Naturally, if the cipher is used to encrypt the same plaintext twice

without changing the key or the spice, the same ciphertext will result.

This has a non-obvious interaction with the method used for encrypting

numbers in a non-power-of-2 range. This could occur when encrypting a

date, which has a range from 0-364. If two different ranges are used

for encryption (or decryption), and the spice and key are unchanged, and

the two ranges require the same number of bits to represent, then the

encryptions will be the same for the two ranges when both plaintext

and ciphertext are in the smaller range. Additional relationships hold

when more plaintext-ciphertext pairs are known in the two ranges.

As a preventive for this situation, the size of the range can be

included in the spice when several ranges are used. This problem is

also prevented by changing the spice for each encryption. The file

hpc.hlp has a brief discussion of this phenomenon at the -leap flag.

Rationale

The expected strength is based on the number of bits of internal state

in the Cipher. The attacker will have to guess most of the internal

state to determine the rest.

Known Attacks

The only known attack is the generic one of guessing some of the

internal cipher state and calculating the rest, trying to relate known

plaintext and ciphertext. Because the cipher has so much internal

state, the attack looks impractical. The irregular structure of the

internal operations makes it difficult to control ``state bloom'' -- a

partial state continually decays, requiring the attacker to make

additional guesses to keep the calculation going.

Depending on the implementation, the cipher will be subject to timing

attacks. This can be countered by changing the spice frequently.

Weak Keys

A key is weak only if its expansion table contains a large number

of low Hamming weight values, or a large number of nearly equal

values. The probability of this occurring for a randomly selected

key is negligible.

Equivalent Keys

Two keys are equivalent if they expand to the same key-expansion

table. The likelihood is negligible for keys of size < 1/2 the

key-expansion table size, 8192 bits. For keys longer than this, some

will be equivalent, but there is no feasible way to discover an

equivalent key pair.

For small blocksizes, only (2^B)! permutations are possible: When

B is 3 bits, there are only 40320 possible permutations, so keys

will be in equivalence classes. (Changing the spice will cause

different encryptions, so the equivalent keys are subdivided into

uniqueness.)

For the small blocksize cipher, Hpc-Tiny, for blocksizes < 36 bits,

an intermediate pseudo-random number is calculated to control the

operation of the cipher. The attacker may be able to restrict the

possible value of this number, if he learns sufficiently many P-C pairs.

For example, for B=5, the intermediate value has 192 bits. Learning

one P-C pair will exclude 31/32 of the possible intermediate values,

leaving only 2^187 possible values. If the entire map is learned, the

number of possible intermediate values is about 2^80. There seems

to be no way to use this information to learn anything about the key,

or the key expansion table. If the key is known, the spice could be

similarly restricted.

Complementation

There are no complementation properties, except for blocksize B = 1 bit.

Key Restrictions

None.

Trapdoors

The Hasty Pudding Cipher contains no trapdoors. All internal constants

and tables are based on the (apparently random) hexadecimal expansions of

well known mathematical constants.

Publications & Analyses

Since Hasty Pudding is new, there are no articles about it.

Advantages & Limitations

The main advantage of the Cipher is flexibility, while retaining good

speed and excellent security. There are things that Hasty Pudding can

do that are impossible for other ciphers. (Read the Overview for

specifics.)

Hasty Pudding offers arbitrary block size, arbitrary key size, flexible

key management, instant key change, and video speeds.

The 64-bit design looks to the future: this cipher will last.

One 64-bit machine, the Alpha, is already in wide use. Intel will be

bringing out their own machine in the near future.

The minimum memory requirement of 2KB may be a challenge for smartcards.

Hasty Pudding will have a larger code+data footprint than many other

ciphers, but memory is cheap and getting cheaper.

Other Applications

The cipher can be used as a hash in various ways, although it has no

speed advantage over SHA or MD5. For example, a keyed hash can be

constructed by placing the object to be hashed in the spice; a file

could be encrypted with itself, or could encrypt a shared-secret value.

Because all the bits are mixed in an encryption, selecting some of them

(perhaps the leading 160) makes a good hash.

The cipher can be used as a MAC generator.

The cipher can be used as a pseudo-random number generator. It has one

interesting advantage as a PRNG: it can jump instantly to any place in

the random number sequence. Most PRNGs can't quickly jump ahead to

the billionth following random number, or go backward. The Hasty Pudding

Cipher can set the seed into the spice, and use a key of length 0.

The Nth random number is calculated by extending N to 128 bits,

encrypting the 128-bit block, and selecting the low 64 bits.

The main drawback to this PRNG is speed: the cost per random number

is higher than typical PRNGs.

The cipher can be used as a stream cipher: each bit or byte can be

separately encrypted. The spice can be incremented after every

encryption. If data dependence of the encryption stream is desired,

the encrypted data items can be numerically added to the spice, or some

portion of the most recent 512 bits of ciphertext can be used in the

spice.

For the longer blocksizes, the cipher is faster than 100 Mbits/sec

on stock hardware (the 300 MHz Alpha).

For ATM, HDTV, B\_ISDN, voice, and satellite applications: The high

bandwidth of the cipher is an advantage. The flexible blocksize is

useful in variable rate applications: there is no need to wait for

a full block of data to accumulate before doing an encryption and

shipping the data. No space is wasted by encrypting; no padding

is required. The spice feature makes parallel encryption easy,

allowing extra hardware to help out when especially high bandwidth

is needed. Another advantage of the flexible blocksize appears

when encrypting parts of headers, while leaving other parts of the

header in the clear. For example, a network firewall connecting

two distant parts of an organization might encrypt the low-order

byte of the IP address to hamper traffic analysis.

Timing Measurements

Encryption speed is independent of key size.

Decryption is about the same as encryption.

There is no specific algorithm setup required.

Key setup time is virtually independent of key size.

Spice change time is instantaneous - 1 instruction.

block 300MHz 250MHz

size Alpha Pentium Pro

1 bit 2.3 usec 14.8 usec

2 3.3

4 3.1

8 7.3 48.8

16 5.5

32 5.0

64 2.1 15.4

79 2.0

128 2.0 13.7

256 2.3

512 2.9 19.4

1024 7.7

2048 15.8

4096 31.9 231.

65537 62.2

1000000 9.2 ms

key setup 79 usec 550 usec

The Pentium is 6-7x as slow as the Alpha.

The Alpha needs 600 clock cycles to encrypt a 128-bit block.

The Pentium appears to need 3500.

Timing Estimates

These are based on extrapolating from the measured data above.

All times are in clock cycles.

Platform: 200 MHz Pentium

keysize/blocksize: 128/128 192/128 256/128

encrypt one data block 3500 3500 3500

decrypt one data block 3500 3500 3500

key setup 140000 140000 140000

algorithm setup 0 0 0

key change 140000 140000 140000

spice change 1 1 1

Platform: 7 MHz Z80 style architecture:

Assume an 8-bit add from memory to memory takes 7 clocks (1 usec).

This works out to 2500 times as slow as an Alpha for 64-bit addition.

The Hasty Pudding Cipher will scale accordingly, needing 5ms to

encrypt a 128 bit block.

keysize/blocksize: 128/128 192/128 256/128

encrypt one data block 35000 35000 35000

decrypt one data block 35000 35000 35000

key setup 1400000 1400000 1400000

algorithm setup 0 0 0

key change 1400000 1400000 1400000

spice change 2 2 2

Platform: 300 MHz Alpha

keysize/blocksize: 128/128 192/128 256/128

encrypt one data block 600 600 600

decrypt one data block 600 600 600

key setup 24000 24000 24000

algorithm setup 0 0 0

key change 24000 24000 24000

spice change 1 1 1

There is no particular time/memory tradeoff: Unrolling loops and

in-lining subroutines gives some speed improvement. Fixing the

blocksize allows some code simplification.

Memory Requirements:

The code footprint is between 10KB and 100KB. The code size is

reduced by using subroutine calls instead of macro expansions,

and not unrolling loops. If some blocksize ranges are not needed,

the code for those ranges can be dropped. (Of course, the

reference implementations include a lot of test and interface code

that would be dropped from a delivery version.)

The key expansion tables require 2300 bytes. If all size ranges

are used, 5 tables are needed, for a volatile memory requirement of

11.5KB. A smart-card could get by with only one key expansion table,

by recomputing the table as needed for each blocksize. A designer

might choose to slightly weaken the cipher and use one key expansion

table for all blocksizes.

In addition, the plaintext or ciphertext block must fit in memory.

(This is not an absolute requirement, but a practical one. The cipher

can encrypt an entire disk or tape as one block, but some intermediate

sorting steps are needed.)

The cipher can encrypt or decrypt a large data block in place, with

only about 100 bytes of extra memory to hold the internal state.

---------------------- hpc.hlp ----------------------

Operating instructions for the Hasty Pudding Cipher.

/\* The Hasty Pudding Cipher \*/

/\* Rich Schroeppel June 1998 \*/

/\* This cipher is in the public domain. \*/

/\* You are free to use it or modify it as you wish. \*/

/\* \*/

/\* Caution: This is experimental code. \*/

/\* The user interface is somewhat confusing, and \*/

/\* you might not be encrypting what you want to, \*/

/\* perhaps with the wrong key. There is no \*/

/\* checking of return codes, so file errors will \*/

/\* go unnoticed. A proper program would erase \*/

/\* the input file. This program hasn't been tested \*/

/\* enough to be sure that an encrypted file will \*/

/\* decrypt properly. The Hasty Pudding Cipher has \*/

/\* not existed long enough to receive significant \*/

/\* cryptographic scrutiny. Your command line input \*/

/\* (including any key that you type) is available \*/

/\* for anyone else on your machine to examine with \*/

/\* ps. \*/

The user interface is a throw-together for hacking with the cipher.

It doesn't check for "user error". There's no checking for file-not-

found etc. (If you are lucky, the program will crash; if you are

unlucky it will encrypt something important in a key known only to

God.)

The Hasty Pudding Cipher is a block cipher. It accepts keys of any

number of bits, and encrypts blocks of any number of bits (even

fractional bits) into blocks of the same size. The no-ciphertext-

expansion feature offers something new in ciphering. There is a

secondary 512-bit key called the \*spice\* which can be changed

instantly, after every encryption if you want. Concealment of the

spice is optional: All or part of it may be revealed, and the cipher

is still secure. The Hasty Pudding Cipher is fast on long data

blocks. The Cipher has a builtin backup mode for extra rounds of

encryption.

I'll begin with an encryption example:

hpc -spi 1 -k x -ia hastypudding -e

35645db6de13e64c 0000000012a9def4

This sets the spice to 1 (seven words of 0 are filled in),

and the key to the single ascii character "x". The phrase

"hastypudding" is encrypted with -e, and the output is printed in hex.

Decrypt it with

hpc -spi 1 -k x -ilen 96 -ix 0x35645db6de13e64c 0x0000000012a9def4 -d

6475707974736168 00000000676e6964

Here I've specified an input length of 96 bits; a length is required for

the -ix option. The output is correct, but hard to read. Adding -txt gives

hpc -spi 1 -k x -ilen 96 -ix 0x35645db6de13e64c 0x0000000012a9def4 -txt -d

h a s t y p u d d i n g

This gives the general flavor. The command looks like

hpc <setup stuff> <action>

<setup stuff> selects the key, spice, input data, and backup & trace modes.

<action> says to encrypt or decrypt, or print some internal state.

setup stuff ---

-spi N ... Sets the spice. Default value is 0.

Follow with 0-8 64bit numbers.

-spi 2 0xff Sets spice[0] to 2, and spice[1] to 255. spice[2...7] are 0.

-b N Sets backup mode. The number is chopped into hex digits.

The low digit sets overall backup mode; the next controls

key-setup; the next five control the subciphers in the order

Tiny, Short, Medium, Long, Xtended. The overall backup value

is added to each of the others.

-b 1 causes everything to have an extra round of encryption.

-b 0x320 Key-setup will use two extra rounds and Tiny mode 3 extra.

-b 0x1001 Short will add two rounds, and everything else one extra round.

-b 0x1112110 same as -b 0x1001.

-t N Sets the trace variable. Individual bits control different

printouts. You will want to use a hex number to select bits.

-t 512 or -t 0x200 both set trace flag 9. (Bit 9 has value 1<<9.)

To set several flags, add the values.

Good -t settings: 0x7c00 for hpc-tiny, 0x21f for hpc-short,

0x276 for hpc-medium and hpc-long, 0x2ff for hpc-xtended.

Program must be compiled with -DTRACE. Not available in the

Java version.

-leap The leap-year flag for -edate.

-txt Cipher output is normally printed as hex numbers. This will

print output as "safe" characters.

-txt -txt prints as raw characters, which will scramble your terminal.

-cbo Turns on the "Cryptix Byte Order" kludge. The program prints

64-bit hex quantities with the bytes reversed. Only these

prints are reversed; others are unchanged. Inputs are not

affected, compounding the kludge. The flag may make it easier

to match up Cryptix files with HPC output. This is a half

measure, since, for example, the kat files will still have

the test items in different orders. Program must be compiled

with -DCBO flag.

-v Prints the program version.

-klen N Chooses a key length (in bits). The key length is normally

determined from the size of the key input, but -klen allows

you to either truncate the key material, or zero pad it.

It's the only way to specify key sizes that are not a multiple

of 4 bits. You might also want to use a long key, but only

specify a short amount of material. Or you might want to use

an initial segment of an input file. -klen should precede

the key material. If you leave out the key, you get zeros.

-k text Uses "text" as the key.

-kx N ... Zero or more 64bit numbers are supplied as the key. Either

decimal or (0x)hex. Requires preceding -klen.

-kxs xxx... Key is one string of hex digits. No 0x prefix.

-kf fname File fname is read in and used as the key.

The Cipher operates internally with 64 bit words. The packing of the key

and input data into those words is of interest:

-k abcdefghij places the ascii code for "a" in the low byte of word 0 of

the key, and the code for "b" in the next byte to the left,

etc. The next word of data begins with "i" in the low byte,

and "j" next.

-kf fname A file is just a long text string.

-kx N N N places the numeric values in successive words.

-kxs xxxxx Blocks of 16 characters are placed into successive key words;

Any leftover is placed right-justified in the last word.

Internally, if the length is not a multiple of 64 bits, the fragment is

defined to be right justified. If the length is not a multiple of 8 (or 4)

bits, the last character of key material is clipped, with high bits ignored.

The input options are similar to the key options.

-ilen Length of the input in bits.

-ia text Input is ascii text.

-ix N ... Zero or more 64bit numbers. Requires preceding -ilen.

-ixs xxxx Single string of hex digits.

-if fname File fname is read in and used as the input.

-o fname Output to file fname. This outputs raw bytes, as you would

want for file encryption. Don't try to encrypt in place -

who knows what will happen? The Cipher doesn't delete or

erase its input file.

-end text Endianness check: Prints text in hex. Must be the only

arguments.

Actions --- one per program invocation.

-pspi Prints the spice.

-pb Prints the backup mode.

-pk Prints the key.

-pka C Expands the key with subcipher C, and prints the expanded

256 word array.

-pi Prints the input.

-foo Prints foo.

-ps Polyscan: computes the swizpoly array. I've already run

this and compiled in the array values, but maybe you want

to compute more.

-ksu N For timing. Runs key setup N times.

-et N L Encrypts an L bit block N times with the Tiny subcipher.

No output - it's for timing. L must be 0-35, or you will

get N errors.

-et2 N L Same as -et, but from code adjacent to Tiny. I-Cache?

-es N L Short cipher. 36<= L <= 64.

-em N L Medium cipher. 65 <= L <= 128.

-em2 Same as -em, but from nearby code.

-el N L Long cipher. 129 <= L <= 512.

-ex N L Xtended cipher. L > 512 bits.

-en N Timing test. Encrypts N times.

-dn N Decryption.

-e Encrypts the input and prints the output. The output is

printed as a bunch of 64bit hex numbers, unless -txt is used.

If -o has selected a filename, that file is written instead,

with straight bytes.

-d Decrypts.

-ebot N Regression test. The input is encrypted for all lengths from

N up to -ilen, and the results printed. Validates algorithm.

-dbot N Decryption.

The next three switches do fractional bit encryption.

-enum N L Encrypts N from the range 0 to L-1 into the same range.

-dnum N L Decrypts.

-edate mmm dd Encrypts the date to another date. mmm should be lower case.

-ddate mmm dd Decrypts.

-leap Sets the leap year flag.

If you play around with date encryption and -leap, you discover a property

of the way number encryption is handled: Changing L from 365 to 366

doesn't affect most encryptions. Everything is the same except that 365

is spliced into the permutation at a random place. If this is a problem,

put the limit L in the spice to get different encryptions. (When L crosses

a power of two, the permutation changes completely.)

In contrast: Changing any bit in the key, or the length, or any spice bit

gives a completely different encryption. Making the input one bit longer,

or changing any bit, even the last bit in a file, gives a completely

different encryption.

-eink Encrypts printable text to printable text. This illustrates

encrypting members of a set, in this case the printable

subset of ascii. Characters in the range from "space" to "~"

are encrypted to the same range. Characters outside the range,

such as tabs, newlines, and meta-characters are passed

unchanged. The spice is incremented after each character, so

even if your text is aaaaaa, the result will look like tr7$@x.

-dink Decrypts.

-nist Out In Runs the NIST required tests. Out is the outer loop count for

the Monte Carlo tests (default 400); In is the inner loop

count (default 10000). Writes seven files. See -cbo.

-lentest S Lb Ib Lm Im Tests consistency of encryption with decryption.

S is the random seed. Lb is the beginning block

length. Lm is the maximum block length. Ib is

the index of the first test. Im is the number of

tests to run at each length. Lm is limited to

1280.

-mct S C Lb Lf E P

Monte Carlo encryption test. S is the random seed, C is the

number of tests to run for each blocksize, Lb is the starting

blocksize, Lf is the final blocksize. E=1 for encryption,

0 for decryption. P=0 to print results only, 1 for summary

only, 2 for both. The output goes into the file mct.txt

unless redirected with -o. The key is always 128 bits.

The key, spice, and plaintext are (pseudo-)random. Prefix

with -b # to test backup mode. The file mct-test tells more.

The command line interface for the standalone Java version is the same,

except that the Java version doesn't have tracing.

Comments & bug reports welcome!

Rich Schroeppel rcs@cs.arizona.edu --revised July 1998

---------------------- hpc-spec ----------------------

Hasty Pudding Cipher Specification

Rich Schroeppel

rcs@cs.arizona.edu

June 1998

Notation & Conventions:

I use C conventions for operators:

+ is addition, - is subtraction, \* is multiplication.

^ is bitwise EXCLUSIVE OR, | is bitwise OR, & is bitwise AND.

>> is right shift, << is left shift.

An operator followed by = means to store the result into the

variable on the left-hand-side. The RHS is completely computed,

and then the op= is computed, and the LHS written.

All numbers are treated as unsigned 64 bit quantities.

Addition and subtraction, and the occasional multiplication

are mod 2^64. Shifts operate on unsigned quantities, with 0 bits

filling empty bit positions.

Bit numbering conventions:

Since all quantities are regarded as 64-bit unsigned integers,

the choice of bit numbering is largely irrelevant.

Bits are numbered from left to right, with bit 63 being the

leftmost bit of a word, and also the numerically largest.

Bit 0 is the rightmost bit, and has the numerical value 2^0 = 1.

If an ascii character string is used as a key, the characters are

placed into an array of 64-bit words, right-to-left. The first

character of the string will occupy bit positions 7-0, the second

character will occupy bit positions 15-8, etc. Within a character,

the bit positions are handled as standard ASCII: An upper case 'A'

has numerical value 65, and ascii bit string 01000001. In this case,

the bits would be numbered as increasing from right to left, with the

low order rightmost bit (a '1' for the character 'A') being bit number 0,

and the leftmost bit (always 0 in standard ascii) being bit number 7.

This usage is consistent with the Hasty Pudding that all variable length

data is represented as arrays with the fragment word at the end of the

array, and the fragment being right justified. The 9 character

string "ABCDEFGHI" would be represented in memory as

word 0

01001000 01000111 01000110 01000101 01000100 01000011 01000010 01000001

word 1

00000000 00000000 00000000 00000000 00000000 00000000 00000000 01001001

When hexadecimal data is presented to Hasty Pudding, a different

convention is used: Complete words are filled in from left to right,

as is any final partial word, but the partial word is right justified.

The hex representation of the 72 bit quantity pictured above is

0x484746454443424149.

The Cryptix convention seems to be different.

Word conventions:

Hasty Pudding accepts data and keys that are not necessarily exact

multiples of 64 bits. The convention used is that such arguments are

supplied as arrays of 64-bit (unsigned) integers. The length in bits

is a separate argument. Any partial word is at the end of the array,

and is right justified. The reference implementation is careful to

ignore any irrelevant high-order bits in the last word, and to

preserve any such bits on output.

Bignum convention:

As defined, Hasty Pudding can encrypt integers, even ones longer than

64 bits. (Although the reference implementation only handles integer

encryptions up to 64 bits.) This is the only case where it's

necessary to define a bignum format. The rule is that the 64-bit

digits of the bignum are placed in an array with the low-order 64 bits

in word 0 of the array, the next 64 bits are placed in word 1, etc.

If there is a partially filled word at the end, the bits are

right-justified.

Introduction

The Hasty Pudding Cipher accepts any size key, of 0 or more bits. The

Cipher accepts any size plaintext block, of 0 or more bits, and

encrypts it without expansion. The Cipher may also be used to encrypt

a range of numbers, or to permute a set of values. Hasty Pudding has

a 512-bit secondary key, the SPICE, for which concealment is optional.

A primary key gives a different encryption for each spice value. To

reduce the cost of cryptographic surprise, the Hasty Pudding Cipher

has a backup option.

Hasty Pudding consists of 5 different sub-ciphers. The blocksize

controls which sub-cipher is used. Each sub-cipher uses its own key

expansion (KX) table, derived from the key.

HPC-Tiny 0 - 35 bits

HPC-Short 36 - 64 bits

HPC-Medium 65 - 128 bits

HPC-Long 129 - 512 bits

HPC-Extended 513+ bits

Internally, the Cipher uses unsigned 64-bit words. Any variable

length value such as a key, a plaintext block, or a ciphertext block,

is supplied as some number of 64-bit words, possibly followed by a

right-justified fragment. The Spice is specified as an array of 8

words. The principal computer operations used are addition and

subtraction, exclusive-or, and shifts. All of these are interpreted

mod 2^64. Shifts are unsigned, with 0 fill.

A few internal "random" numbers are used in the cipher.

PI19 = 3141592653589793238

E19 = 2718281828459045235

R220 = 14142135623730950488

Key Expansion (KX) Tables

Each subcipher has a KX (key expansion) table, 256 words of 64-bits,

pseudo-randomly generated from the key. All five tables may be

computed when a key is setup, or the tables may be computed when

needed. An application which only used a few blocksizes would need

only a subset of the tables. The same algorithm is used for each KX

table, changing only an initialization. The KX tables are firewalled:

knowing a KX table won't help find the original key, or a KX table for

a different subcipher.

Each KX table is followed by 30 words which are a copy of the first 30

words. This allows the cipher to reference the tables as if the index

were wrapped mod 256.

Creating a KX table:

The parameters are

the sub-cipher number (from 1 to 5, 1 is HPC-Tiny);

the key length in bits (a non-negative integer);

and a pointer to an array containing the key bits.

If the key length is not a multiple of 64 bits, the fragment is

right-justified in the last word of the array.

The first three words of the KX array are initialized:

KX[0] = PI19 + sub-cipher number.

KX[1] = E19 \* the key length.

KX[2] = R220 rotated left by the sub-cipher number of bits.

The remaining 253 words of the array are pseudo-randomly filled in

with the equation

KX[i] = KX[i-1] + (KX[i-2] ^ KX[i-3]>>23 ^ KX[i-3]<<41).

Then the key is xored into the KX array. Word 0 of the key is xored

into word 0 of the KX array, etc. If there is a key fragment, unused

high-order bits are masked to 0 before doing the xor. When the last

of the key has been xored into the KX array, the Stirring function is

run to mix up the bits.

For very long keys: After 128 words of key have been xored into the KX

array, the Stirring function is run. If there is more key to use, the

next 128 words are xored into the stirred KX array, starting over at

word 0 of KX. (No key is ever xored directly into the second half of

the KX array, but these bits are affected by the stirring function.)

Boundary cases: The Stirring function is always run at least once,

even if the key length is 0. If the key is an exact number of

128-word blocks, the Stirring function is only done once (not twice)

after xoring in the last chunk of key.

The Stirring function:

The purpose of the Stirring function is to pseudo-randomize the KX

array, allowing each bit to influence every other bit.

The function does several passes over the KX array, altering every

word. The default number of passes is 3. The backup feature causes

additional passes. The number of extra passes is the sum of the

global backup variable BACKUP and the array entry BACKUPSUBCIPHER[0].

Normally both variables are 0.

The Stirring function has 8 internal state variables, each an unsigned

64bit word. They are called s0...s7 below. Before the first pass

over the KX array, they are initialized from the last 8 values in the

array.

s0 = KX[248], ..., s7 = KX[255].

One pass over the KX array: Each word in the KX array is mixed with

the state variables, starting with KX[0] and working through KX[255].

Each array word is overwritten after mixing. The mixing function is

deliberately made slightly lossy, so that the process cannot be run

backward to discover the pre-stirred KX value, and hence the key.

The individual word stirring function is specified by the code below:

s0 ^= (KX[i] ^ KX[(i+83)&255]) + KX[s0&255] /\* lossy, sometimes \*/

s1 += s0

s3 ^= s2

s5 -= s4

s7 ^= s6

s3 += s0>>13

s4 ^= s1<<11

s5 ^= s3<<(s1&31)

s6 += s2>>17

s7 |= s3+s4 /\* lossy \*/

s2 -= s5 /\* cross-link \*/

s0 -= s6^i

s1 ^= s5 + PI19

s2 += s7>>j

s2 ^= s1

s4 -= s3

s6 ^= s5

s0 += s7

KX[i] = s2+s6

I is the index of the array word being mixed, and varies from 0 to 255.

J is the pass number, starting at 0 for the first pass.

Finishing up Key Expansion

After the entire key has been xored into the KX array, and the

last stirring of the array, the first 30 words of the array are copied

onto the end.

KX[256] = KX[0], KX[257] = KX[1], ..., KX[285] = KX[29].

The purpose is to allow code that references the KX array to wrap-around

array indexes mod 256, without having to mask the index to the low-order

8 bits.

Spice

The SPICE is a secondary key of 512 bits. At the option of the

application designer, it may be completely or partially concealed, or

completely open. It may be implicit, automatically set from the date,

or the filename, or disk block number. Since the spice is cheap to

change, I expect it will be changed often, perhaps for every encrypted

block. This allows the primary key to have a long lifetime, perhaps

years, since the amount of material encrypted in any single key+spice

combination is small.

As long as the key is secret, the spice can be kept secret: If the

attacker knows a lot of plaintext- ciphertext pairs, but does not know

the key, he can't determine the spice. If he knows part of the spice,

he can't learn any of the rest. Since he doesn't know the key, even

exhaustive search of the spice space is impossible.

Caution: Avoid designing systems that let a potential opponent control

the spice value. The cipher might be vulnerable to a "Chosen Spice"

attack. In such an attack, an opponent would have access to

plaintext-ciphertext pairs, and manipulate the spice (while the key

remained unchanged) to see how the plaintext or ciphertext was altered

in subsequent encryptions. [I don't know of such an attack that

works, and have tried to design against this type of attack, but this

is an area that needs crypto community analysis.]

The SPICE is an array of 8 64-bit words. Its default value is all 0s.

The encrypting application may set any value into the spice. The

decrypting application must use a matching value to correctly decrypt.

For efficiency reasons, a designer may choose to use a truncated

spice, or to omit the spice completely from an implementation. The

cipher is as secure as ever. (Ultra-cautious designers will want to

review the lifetime of the key.) The unused spice portion is deemed

to have the value 0. Shortened spice implementations will

interoperate with those that use the full spice, or a longer portion,

so long as the more capable version takes care to 0 the unused part of

the spice.

The raw Cipher does not write into the spice, but I have supplied

a spice-incrementing encryption mode in the API. This has some of

the virtues of CBC mode, but allows encryption in parallel.

The Backup Option

The Hasty Pudding Cipher includes a backup option. This helps to

limits the damage from cryptographic surprise. If the backup option

is activated, the cipher does extra mixing steps, making it harder to

break. Since the backup option is always available, it won't be

necessary to deploy a new encryption method under emergency

conditions.

There is a backup variable, BACKUP, and an array, BACKUPSUBCIPHER[].

The array contains one number for each subcipher. (The key-setup

stirring function uses word 0 of the array.) The numbers specify the

amount of extra work that each subcipher (or the key setup function)

does. In normal operation, all the backup parameters are 0,

specifying no extra work. Backup mode is activated by setting any

parameter to a positive value. Each subcipher (and the key-setup

function) adds its backup variable to the global BACKUP value to

determine the amount of extra work. The global variable activates

extra security with one switch. The array parameters allow

fine-grained control, so that the security level can be adjusted for

individual subciphers. Setting BACKUP to 1 is exactly equivalent to

setting all the other backup parameters to 1.

The execution time cost of setting the backup option: For individual

subciphers, setting the backup level to N will cause N extra

encryptions, slowing the cipher by a factor of N+1. For key setup,

setting the backup level to N will add N extra rounds of stirring to a

baseline of 3, so the slowdown will be a factor of 1 + (N/3).

The backup option does multiple encryptions, using the same key and

spice combination as single encryption. The cycle number of the

encryption is added to the intermediate ciphertext before each

encryption cycle. The addition is to word 0 of the ciphertext, taken

mod 2^64; no carry is propagated. Short ciphertext blocks (less than

64 bits) are remasked to the correct length, dropping any carry bit.

The original encryption is cycle number 0; extra encryptions are

cycles 1, 2, etc.

HPC-Tiny, the routine for encrypting tiny blocks of <36 bits, makes

calls to HPC-Medium and HPC-Long. These recursive calls are not

affected by the backup function; the ciphers don't do additional work.

HPC-Tiny has an additional backup wrinkle, explained at its

description.

The backup option is implemented in the top-level routine that handles

the selection of the subcipher - individual subciphers are unaware of

the backup option, except for HPC-Tiny. The intermediate ciphertext

values are stored in the output array during the extra encryptions.

The SubCiphers

All of the subciphers move the target plaintext into intermediate

variables, named s0...s7. On a register-rich machine, these variables

will be in the registers. The intermediate variables are all unsigned

64-bit words. Most blocksizes will have one leftover fragment. This

fragment is operated on much like the other variables. The fragment

is kept right-adjusted. Anytime the fragment is operated on, the

result is masked to the correct fragment length. Anytime the fragment

is used as an operand, it is masked before use. (This description is

conceptual - the reference implementation omits much redundant

masking.) A mask variable, LMASK, is kept handy for the masking

operations. It is a right-justified block of 1s. When the blocksize

is a multiple of 64, the subciphers regard the fragment as 64 bits,

rather than 0. (This provides an optimization opportunity for the AES

target blocksize of 128 bits.) If the blocksize is divisible by 64,

the mask is all 1s. If the blocksize is 11 (mod 64), the mask is

0x7FF. The mask is never 0.

Each subcipher works by stirring the state variables repeatedly. Data

from the key and spice is mixed in at strategic times. Each

individual stirring step is reversible. Decryption is done by

reversing the steps. In most cases the appropriate reversal is

obvious, but a couple of complicated cases are explained below.

Each cipher begins by adding some of the expanded key to the

plaintext, and finishes off by adding more expanded key to create the

final output ciphertext. The words from the key are selected based on

the blocksize. If the blocksize is B bits, then word B from the KX

array is added to plaintext word 0 to initialize state variable s0.

KX[B+1] is added to plaintext word 1 to initialize state variable s1.

Etc. The final sum (probably a fragment) is masked. Then the

subscipher specific subscipher mixing algorithm is run. Finally, just

before output, word B+8 of the KX array is added to state variable s0

to create word 0 of ciphertext; KX[B+9] is added to state variable s1

to create word 1 of ciphertext, etc. The fragment is correctly

masked. Only the masked portion of the last ciphertext word is

changed. The reference implementation is coded to allow in-place

encryption, where the ciphertext array is the same as the plaintext

array. (Overlapping doesn't work.)

HPC-Medium blocksize 65 to 128 bits

The plaintext is copied to cipher variables s0 and s1. S1 contains

the right-justified fragment. A mask is calculated for clipping S1 to

the fragment length before and after use. (The masking operations are

omitted from the description.) Two consecutive words from the KX

array are added to s0 and s1. KX[blocksize] is added to s0.

The intermediate mixing consists of 8 rounds, numbered 0-7. The

variable I in the code below is the round number. The round algorithm

is

k = KX[s0&255]

s1 += k

s0 ^= k<<8

s1 ^= s0

s1 &= lmask Fragment masking example. Other uses omitted.

s0 -= s1>>11

s0 ^= s1<<2

s0 -= spice[i^4]

s0 += (s0<<32) ^ (PI19 + blocksize)

s0 ^= s0>>17; s0 ^= s0>>34

t = spice[i]

s0 ^= t

s0 += t<<5

t >>= 4

s1 += t

s0 ^= t

s0 += s0 << (22 + (s0&31))

s0 ^= s0>>23

s0 -= spice[i^7]

t = s0&255

k = KX[t]

kk = KX[t+3\*i+1]

s1 ^= k

s0 ^= kk<<8

kk ^= k

s1 += kk>>5

s0 -= kk<<12

s0 ^= kk &~ 255

s1 += s0

s0 += s1<<3

s0 ^= spice[i^2]

s0 += KX[blocksize+16+i]

s0 += s0<<22

s0 ^= s1>>4

s0 += spice[i^1]

s0 ^= s0>>(33+i)

After completing the 8 rounds, KX[blocksixe+8] is added to s0, and the

next KX word to s1. s0 and s1 are stored into the output. s1 is

masked, and any high-order bits (to the left of the mask) in the final

word of the output array are not changed.

Decryption

Decryption is done by reversing the encryption steps. The order of

the steps is (generally speaking) reversed, and the individual steps

are replaced by their inverses. The round number is counted from 7

down to 0. A step such as s0 += s1 is reversed to s0 -= s1, while a

step like s0 ^= s1 is self-inverse. The masking steps are not exactly

reversed, but must be carefully inserted after the main reversal is

worked out.

Some special cases:

t = spice[i]; s0 ^= t

In a sequence such as this, the value of t must be loaded from

spice[i] before use, even in the decryption routine. So the sequence

is self-inverse.

k = KX[s0&255]; s1 += k; s0 ^= k<<8

The low order 8 bits of s0 are unchanged by this sequence, so k =

KX[s0&255] will pick up the same word from the KX array on both

encryption and decryption passes.

s0 ^= s0>>23

The inverse for this is s0 ^= s0>>23; s0 ^= s0>>46.

The theme works as long as the amount shifted is at least a

quarter-word (16 bits). This and the following example will also work

for left shifts. The combining operator must be xor.

s0 ^= s0>>17; s0 ^= s0>>34

The inverse for this is just s0 ^= s0>>17.

s0 += (s0<<32) ^ (PI19 + blocksize)

This is reversed by doing the reverse twice:

t = s0 - ((s0<<32) ^ (PI19 + blocksize)) low order 32 bits correct.

s0 -= (t<<32) ^ (PI19 + blocksize) low order 64 bits, i.e. all.

s0 += s0 << (22 + (s0&31))

There are two points here: First, the low 5 bits of s0 are undisturbed

by the += addition operation, since the addend has at least 22

low-order 0 bits. So s0&31 will calculate the same thing in the

reversed code as in the forward code. The second point is that the

addition is reversed by duplicating the inverse of the forward

operation:

t = 22 + (s0&31) Calculate the shift amount.

u = s0 - (s0<<t) The low order 2t bits of u are a correct reversal.

s0 -= u<<t The low order 3t bits of s0 are correctly reversed.

This reversal works as long as the amount shifted exceeds a third of a

word, 21.333 bits. T is at least 22.

An add-to-self of a right shifted self, such as s0 += s0>>50, is not

invertible.

HPC-Long 129 to 512 bits

This subcipher has 8 state variables, s0...s7. The plaintext is

copied to cipher variables s0,s1,...,s7. S0 gets the 0 word of the

plaintext, s1 the next word, etc. The fragment is always placed in

s7. The other variables are used in order. S0 and s1 are always

used; s2 is used only when the blocksize exceeds 192, etc. The

minimal blocksize for this subcipher, 129 bits, uses only s0, s1, and

the low-order bit of s7. The maximum blocksize, 512 bits, uses all

bits of all 8 state variables.

A mask is calculated for clipping S7 to the fragment length before and

after use. (The masking operations are omitted from the description.)

Up to 8 consecutive words from the KX array are added to s0...s7:

KX[blocksize&255] is added to s0, the next KX word to s1, etc.

For the smaller blocksizes, the additions can be skipped for s2...s6

when the variables are not used. Regardless of the blocksize,

KX[(blocksize&255)+7] is always added to s7 (which is then masked).

The mixing function is similar to HPC-Medium, with a few wrinkles.

There are 8 rounds of mixing, described below. I is the round number,

varying from 0 to 7.

t = s0&255

k = KX[t]

kk = KX[t+3\*i+1]

s1 += k

s0 ^= kk<<8

kk ^= k

s1 += kk>>5

s0 -= kk<<12

s7 += kk

s7 ^= s0

s1 += s7

s1 ^= s7<<13

s0 -= s7>>11

s0 += spice[i]

s1 ^= spice[i^1]

s0 += s1<<(9+i)

s1 += (s0>>3) ^ (PI19+blocksize)

s0 ^= s1>>4

s0 += spice[i^2]

t = spice[i^4]

s1 += t

s1 ^= t>>3

s1 -= t<<5

s0 ^= s1

The next part of the mixing function depends on the blocksize.

This is so that only occupied variables get mixed.

If the blocksize exceeds

448: s6 += s0; s6 ^= s3<<11; s1 += s6>>13; s6 += s5<<7; s4 ^= s6;

384: s5 ^= s1; s5 += s4<<15; s0 -= s5>>7; s5 ^= s3>>9; s2 ^= s5;

320: s4 -= s2; s4 ^= s1>>10; s0 ^= s4<<3; s4 -= s2<<6; s3 += s4;

256: s3 ^= s2; s3 -= s0>>7; s2 ^= s3<<15; s3 ^= s1<<5; s1 += s3;

192: s2 ^= s1; s2 += s0<<13; s1 -= s2>>5; s2 -= s1>>8; s0 ^= s2;

(Longer blocksizes also execute the code for shorter blocksizes.)

s1 ^= KX[(blocksize + 17 + (i<<5))&255]

s1 += s0<<19

s0 -= s1>>27

s1 ^= spice[i^7]

s7 -= s1

s0 += s1 & s1>>5

s1 ^= s0 >> (s0 & 31)

s0 ^= KX[s1&255]

After completing the 8 rounds, KX[blocksixe+8] is added to s0, and the

next 7 KX words to s1...s7. KX[blocksize+15] is always added to s7,

regardless of any unused variables among s2...s6. S0 and s1 are

stored into the output. When the blocksize warrants, some or all of

s2...s6 are stored into the output. S7 is masked and stored into the

final output word. Any high-order bits (to the left of the mask) in

the final word of the output array are not changed.

Decryption is done by reversing the encryption steps. The round

number is counted down from 7 to 0. See the discussion about reversal

after HPC-Medium for examples.

HPC-Extended blocksize 513 bits and larger

This subcipher has 8 state variables, s0...s7. Since the blocksize

exceeds the capacity of the state variables, a new strategy is

necessary.

The first 7 of the state variables, s0...s6, are used in the ordinary

way. But s7 is a "swapping" state variable: Most of the plaintext is

swapped (one word at a time) into s7 for mixing, and then swapped back

out. The plaintext is processed this way (every word swapped in,

mixed, swapped out) three times. This gives every plaintext bit a

chance to affect every ciphertext bit. Changing any bit, even the

final bit, of a long block of plaintext will (on average) randomly

flip half the bits in the resulting ciphertext.

The details:

At the start of the cipher, several calculations are made for later

use. These are based on the blocksize to be encrypted.

LWD is the number of input words, blocksize/64, rounded up if there's

a fragment.

LMASK is a right-justified mask of 1s for the fragment. It always

contains at least 1 1bit.

QMSK is one less than the smallest power of 2 >= LWD.

SWZ is the smallest Swizpoly number that exceeds QMSK.

Swizpoly numbers:

0, 3, 7, 0xb, 0x13, 0x25, 0x43, 0x83, 0x11d, 0x211, 0x409,

0x805, 0x1053, 0x201b, 0x402b, 0x8003, 0x1002d, 0x20009,

0x40027, 0x80027, 0x100009, 0x200005, 0x400003, 0x800021,

0x100001b, 0x2000009, 0x4000047, 0x8000027, 0x10000009,

0x20000005, 0x40000053, 0x80000009

These will be explained below.

For a blocksize of 513 bits, LWD=9, LMASK=1, QMSK=15, SWZ=0x13=19.

For a blocksize of 1024 bits, LWD=16, LMASK = 0xffffffffffffffff (all

1s), while QMSK and SWZ are unchanged. If we nudge the blocksize up

to 1025 bits, LWD=17, LMASK falls back to 1, QMSK jumps to 31, and SWZ

becomes 0x25=37.

LMASK is handled in a more complicated way than in the other

subciphers. As words of plaintext are swapped into s7 for mixing,

LMASK always corresponds to the number of valid bits in s7. This

means it is usually all 1s, except when the last word of the plaintext

is being processed.

Next we are ready to initialize the state variables: s0...s7 are

copied from the first 8 words of plaintext, and 8 words are added from

the KX array.

s0 = ptx[0] + KX[blocksize&255]

s1 = ptx[1] + the next word of the KX array

etc.

s7 = ptx[7] + KX[...+7]

Now we begin the mixing operation. The Stir function is defined

below. It operates on s0...s7, with various additional inputs,

including the round-index I. For now, we will simply assume it is

defined, so we can get on with the higher level description.

LMASK is set to all 1s, corresponding to s7 containing 64 bits of

plaintext from ptx[7]. A premixing step is run: Stir is called 4

times, with round-index I = 0...3. s7 is written out to word 7 of the

ciphertext array, to make room for swapping. The round-index I is set

back to 0. Starting with plaintext word 8, ptx[8], each word of

plaintext is brought into s7. Nothing from the KX array is added.

LMASK is set to match the number of valid bits in s7: it remains all

1s until the last word of plaintext is copied into s7; then it is

recalculated from the fragment size, and s7 is masked. The Stir

function is run once. s7 is written back to the ciphertext (output)

array, in the same relative position as the swapped-in plaintext

plaintext word. When the final plaintext word is processed, care is

taken to only write the valid bits of the fragment into the

ciphertext, leaving any high-order bits unchanged. This is the end of

the first pass over the plaintext. The partially processed data now

resides in the output block, except for words 0-6 which are retained

in s0...s6.

Now we have a brief intermission to prepare for the second pass over

the data. After the last plaintext word is processed and written out,

s7 is reloaded from ciphertext word 7, LMASK is reset to all 1s, and

the Stir function is run again. The blocksize is added to s0. The

Stir function is called 3 more times, with the round-index going from

0 to 2. The blocksize is added to s0 again. The round-index is reset

to 0, and s7 is written back to ctx[7]. We are now ready for the

second pass over the data.

The data words are again swapped, one at a time, into s7. However,

there are two differences. First, the data is now brought in from the

ciphertext array where it was temporarily stored.

[We don't want to stomp on the plaintext, but we are implicitly

allowed to stomp on the output array where the final ciphertext will

go. This brings up an incidental side issue: The reference

implementation allows for encryption in-place, with the ciphertext

overwriting the plaintext. For this to work properly, only one word

at a time can be swapped in from the plaintext array, because of the

next point.]

Second, the data words from the ciphertext array are processed in a

different order for this (the second) pass. We will postpone defining

the "picking order" for this pass. Suffice it to say that each

ciphertext word, from ctx[8] through the end at ctx[LWD-1], is swapped

into s7 once, processed, and written back to the same position in the

ctx array. The "last" word, ctx[LWD-1], which may contain a fragment,

is processed somewhere in the middle. As each word is swapped in,

LMASK is set appropriately: to all 1s except when the fragment word is

swapped in, otherwise to the appropriate block of 1s. When the

fragment is written back out, the code is careful not to disturb any

unused high-order bits. The processing for each word is to run the

Stir function once. (The round-index is 0 for the entire pass.)

After all the data words have been processed and written into the

output array, there is an intermission to prepare for the third pass.

s7 is reloaded from ctx[7], and LMASK is set to all 1s. The

round-index is set to 1, and the Stir function is run. The blocksize

is added to s0 another time. The Stir function is run three more

time, with the round-index varying from 0...2. The blocksize is added

to s0 once more. s7 is stored into ctx[7] again. The round-index is

reset to 0, and we are ready for the final pass over the data.

The third pass is the same as the second, with the data words swapped,

one at a time, from the ctx array into s7, LMASK set, the Stir

function run once, and s7 written back into the ctx array in the same

location. The third pass uses a different picking order than either

of the first two passes.

After all the data has been processed and written out the third time,

we are ready for the finale. s7 is loaded from ctx[7]; LMASK is set

to all 1s; the round index is still 0. The Stir function is run once.

Then it is run 3 more times, with the round-index varying from 0...2.

Finally, 8 words of the KX array are added to s0...s7, and the sums

written into ctx[0]...ctx[7].

ctx[0] = s0 + KX[(blocksize&255)+8]

ctx[1] = s1 + KX[(blocksize&255)+9]

etc.

ctx[7] = s7 + KX[(blocksize&255)+15]

and we are done!

It remains to define the function Stir, and the data picking order for

passes two and three.

Stir is similar to HPC-Long. I is the round-index. S7 must be masked

before each use, and when swapped in or out.

t = s0 & 255

k = KX[t]

kk = KX[t+1+(i<<2)]

s3 += s7

s5 ^= s7

s1 += k

s2 ^= k

s4 += kk

s6 ^= kk

s4 ^= s1

s5 += s2

s0 ^= s5>>13

s1 -= s6>>22

s2 ^= s7<<7

s7 ^= s6<<9

s7 += s0

t = s1 & 31

tt = s1>>t

ttt = s2<<t

s3 += ttt

s4 -= s0

s5 ^= ttt

s6 ^= tt

s7 += tt

t = s4>>t

s2 -= t

s5 += t

If the round-index I is 1, then the Spice is combined with s0...s7:

s0 += spice[0]; s1 ^= spice[1]; s2 -= spice[2]; s3 ^=spice[3]

s4 += spice[4]; s5 ^= spice[5]; s6 -= spice[6]; s7 ^=spice[7]

s7 -= s3

s1 ^= s7>>11

s6 += s3

s0 ^= s6

The next sequence exchanges the even-numbered bit positions in s2 and s5.

S3 and s0 are modified as targets of opportunity.

t = s2^s5

s3 -= t

t &= 0x5555555555555555

s2 ^= t; s5 ^= t

s0 += t

t = s4<<9

s6 -= t

s1 += t

Next we define the data word picking order for passes two and three.

Two different iterations methods are defined. Each goes through all

N-bit values, in pseudo-random order. N is log-base-2 of QMSK+1. We

have calculated QMSK above, so that QMSK+1 is the smallest power of 2

as big as LWD. The first iteration method, used for pass two, is

Qnew = 5\*Q + 1, followed by Qnew &= QMSK.

It begins and ends with Q=0. Q is masked by QMSK after each step, to

restrict it to N-bit values. During the course of the iteration, when

Q takes on values less than 8 or greater than LWD, it is simply

stepped again. When Q takes values in the range [8,LWD], then word

ctx[Q] is swapped into s7, mixed, and swapped out. (Before ctx[Q] is

processed, a check for Q=LWD is made to compute LMASK.) It's an easy

theorem that the iteration steps through all N-bit values exactly once

before returning to 0.

The second iteration method is used for pass three. QMSK is

incremented to become an exact power of 2, a one-bit mask.

Qnew <<= 1; if (Qnew & QMSK) then Qnew ^= SWZ;

The iteration begins and ends with Q=1. Q takes on every N-bit

value except for Q=0, which we don't need anyway. As in pass two,

whenever Q is in the range [8,LWD], we swap ctx[Q] into s7 for mixing,

and swap it back out. The SWZ values are taken from the Swizpoly

numbers: each Swizpoly entry corresponds to the (lexicographically)

first GF[2] polynomial of degree N for which X is a primitive root.

If this sounds like gobbledygook, then Swizpoly may be defined

empirically as the smallest number > 2^N for which the above iteration

takes on all 2^N-1 nonzero values.

Decryption

Decryption works by reversing the encryption steps, just as for the

other subciphers. The round-index must be counted backward, since it

is used as an input to the Stir function. There are a couple of fine

points. Toward the end of the definition of Stir, there is a sequence

of instructions that exchanges the even-numbered bit positions of s2

and s5.

t = s2^s5

s3 -= t

t &= 0x5555555555555555

s2 ^= t; s5 ^= t

s0 += t

This is inverted by

t = s2^s5

s3 += t

t &= 0x5555555555555555

s2 ^= t; s5 ^= t

s0 -= t

Since the bits of s2 and s5 were exchanged, the decryption value for t

is the same as the encryption value.

The picking-order iterations for passes two and three must be run

backwards. The end values are easy enough, matching the starting

values.

The pass two inverse iteration is Q = (Q-1)/5 (mod 2^N). This looks

messy, using a division - a modular one, yet. However, the situation

is saved when we notice that the 2-adic reciprocal of 5 is ...ccccccd

in hex. This means we can use the formula

Qold = (Q-1) \* 0xcccc...cccd (mod 2^N)

The multiplication can be implemented with a few shifts & adds:

Q = Q-1

QQ = Q<<2; QQ += QQ<<1; QQ += QQ<<4; QQ += QQ<<8; QQ += QQ<<16

Qold = (Q+QQ) & QMSK;

The inverse iteration for pass three is easier:

if (Q & 1) then Q ^= SWZ SWZ is odd, zeroing the low bit of Q.

Q >>= 1

This closes off the loose ends for the HPC-Extended subcipher.

HPC-Short blocksize 36 to 64 bits

The plaintext is placed right-justified in variable s0. LMASK is set

to a block of 1s, to mask s0 to the valid bits between operations.

The masking operations are omitted from the description.

A word from the KX array, KX[blocksize], is added to s0.

Several shift sizes are calculated:

LBH = (blocksize+1)/2 Division rounds down.

LBQ = (LBH+1)/2

LBT = (blocksize+LBQ)/4 + 2

GAP = 64 - blocksize

Then 8 rounds of mixing are run, with round-index I going from 0...7.

After the mixing, another word from KX, KX[blocksize+8] is added to

s0. S0 is masked, and the valid bits are written to the output array.

Any high-order bits in the output array are unchanged.

The mixing function is

k = KX[s0&255] + spice[i]

s0 += k<<8

s0 ^= (k>>GAP) &~255

s0 += s0<<(LBH+i)

t = spice[i^7]

s0 ^= t

s0 -= t>>(GAP+i)

s0 += t>>13

s0 ^= s0>>LBH

t = s0&255

k = KX[t]

k ^= spice[i^4]

k = KX[t+3\*i+1] + (k>>23) + (k<<41)

s0 ^= k<<8

s0 -= (k>>GAP) &~255

s0 -= s0<<LBH

t = spice[i^1] ^ (PI19+blocksize)

s0 += t<<3

s0 ^= t>>(GAP+2)

s0 -= t

s0 ^= s0>>LBQ

s0 += Permb[s0&15]

t = spice[i^2]

s0 ^= t>>(GAP+4)

s0 += s0<<(LBT + (s0&15))

s0 += t

s0 ^= s0>>LBH

The array Permb is chosen so that adding an entry indexed by the low 4

bits permutes the 4 bit values. This allows an analogous subtraction

operation to invert the operation for decryption. In contrast to

Perma, all 64 bits of the data for Permb and Permbi are important.

Permb[16]:

0xB7E151628AED2A6A -0, 0xBF7158809CF4F3C7 -1,

0x62E7160F38B4DA56 -2, 0xA784D9045190CFEF -3,

0x324E7738926CFBE5 -4, 0xF4BF8D8D8C31D763 -5,

0xDA06C80ABB1185EB -6, 0x4F7C7B5757F59584 -7,

0x90CFD47D7C19BB42 -8, 0x158D9554F7B46BCE -9,

0x8A9A276BCFBFA1C8 -10, 0xE5AB6ADD835FD1A0 -11,

0x86D1BF275B9B241D -12, 0xF0D3D37BE67008E1 -13,

0x0FF8EC6D31BEB5CC -14, 0xEB64749A47DFDFB9 -15.

Permbi[16]:

0xE5AB6ADD835FD1A0 -11, 0xF0D3D37BE67008E1 -13,

0x90CFD47D7C19BB42 -8, 0xF4BF8D8D8C31D763 -5,

0x4F7C7B5757F59584 -7, 0x324E7738926CFBE5 -4,

0x62E7160F38B4DA56 -2, 0xBF7158809CF4F3C7 -1,

0x8A9A276BCFBFA1C8 -10, 0xEB64749A47DFDFB9 -15,

0xB7E151628AED2A6A -0, 0xDA06C80ABB1185EB -6,

0x0FF8EC6D31BEB5CC -14, 0x86D1BF275B9B241D -12,

0x158D9554F7B46BCE -9, 0xA784D9045190CFEF -3.

Permb was derived from the hex expansion of e (2.718...). The

fraction was grouped into chunks of 64 bits, and the first sixteen

chunks with unique low-order 4bit hex digits were selected. The

twelfth and fourteenth entries would have been fixed points for the

low-order 4 bits, so they were swapped.

Decryption

As usual, decryption is done by reversing the encryption operations.

Fine points (see discussion for decrypting HPC-Medium).

The inverse for s0 ^= s0>>LBQ is s0 ^= s0>>LBQ; s0 ^= s0>>(2\*LBQ).

The inverse for s0 += s0<<(LBT + (s0&15)) is

t = lbt + (s0&15); s0 -= (s0 - (s0<<t)) << t.

The inverse for s0 += Permb[s0&15] is s0 -= Permbi[s0&15].

HPC-Tiny blocksize 0 to 35 bits

Encryption of a 0-length block is a nop, and returns immediately.

The HPC-Tiny subcipher faces a new kind of challenge: Involving enough

of the key expansion array and the spice to make a meaningful

encryption. For very small blocksizes, the attacker has an

interesting new option available: searching spice-space, getting

complete plaintext-ciphertext maps, and trying to make deductions from

spices that produce identical or nearly identical P-C maps.

HPC-Tiny is subdivided into several subsubciphers, for different

blocksizes. One key expansion array serves for all. To involve all

of the spice, and enough of the KX array, a (one-level) recursive call

is made to either HPC-Medium or HPC-Long to create a pseudo-random

number that depends on the necessary key and spice inputs. This

number then controls the permutation applied to the plaintext. The

number provides an additional point of attack: we must usually prevent

the attacker from learning its value. The decryption code also

computes this same number, so HPC-Tiny decryptions include

\*ENcryption\* calls to HPC-Medium or HPC-Long.

The backup option provides some additional complexity: First, the

backup round number is passed in the blocksize argument. The real

blocksize is in the low order six bits; the other bits are right

shifted 6 to recover the backup round number. (When the cipher is

running in non-backup mode, it's the same as backup round 0. Backup

mode also uses round 0 as the start for multiple encryptions.)

Immediately prior to the recursive calls that HPC-Tiny makes to

HPC-Medium or HPC-Long, the backup round number is added to word 0 of

the temporary plaintext array. The resulting ciphertext will be

completely different on each round, providing different input to the

permutation engine.

For all of the subsubciphers, encryption begins with the same pattern

used in the other subciphers: The plaintext is placed right-justified

in variable s0. LMASK is set to a block of 1s, to mask s0 to the

valid bits between operations. As usual, the masking operations are

omitted from the description.

A word from the KX array, KX[blocksize], is added to s0 (which is then

masked).

The subsubcipher dispatch is made, based on the blocksize. When it

returns, the usual output steps are followed:

Another word form the KX array, KX[blocksize+8] is added to s0. S0 is

masked, and the valid bits are written into word 0 of the output

array, leaving the high-order bits of the output word unchanged.

The HPC-Tiny Subsubciphers

The subsubciphers for blocksizes 1-4 all begin with a call to

HPC-medium. A two-word temporary array tmp[] is copied from

KX[16+2\*blocksize] and KX[17+2\*blocksize]. This is encrypted in-place

as a 128 bit block by HPC-Medium, using the same KX array, and the

same spice. The resulting pseudo-random array of two words controls

the permutation applied to s0.

For a 1-bit blocksize, the pseudo-random number is be distilled down

to a single bit, which is xored with s0. A 128 bit number N is

created, with the low 64 bits as the sum of tmp[0] and tmp[1], and the

high 64 bits as tmp[1]. The number N is subjected to a process I call

Fibonacci Folding to condense it down to a single bit:

N += N>>89; N ^= N>>55

N += N>>34; N ^= N>>21

N += N>>13; N ^= N>>8

N += N>>5; N ^= N>>3

N += N>>2; N ^= N>>1; N += N>>1;

The low-order bit of N is xored into s0.

For this case, decryption is identical to encryption.

For blocksizes 2 and 3, each word of the tmp array is used to control

a simple permutation engine that operates on s0. Tmp[0] is used

first. The low order blocksize bits are xored into s0, and the next

blocksize bits are added to s0. S0 is then rotated 1 bit left, and

the used 2\*blocksize bits of the temporary variable are right shifted

away. Enough cycles are done (16 or 11) to entirely use each

variable.

Decryption is a straightforward reversal of the steps.

Note that each of the steps described is either an even or an odd

permutation on the space of 2^blocksize numbers. Xoring is even;

addition is odd or even as the low-order bit added is 1 or 0; rotation

is even for blocksizes other than 2. An attacker can learn something

about the parity of certain bits in the tmp[] array, if he knows the

entire plaintext-ciphertext map. (I believe that this knowledge is

not useful for a further attack.)

For 4 bit numbers, the processing is a little more interesting. Each

word of the tmp array is used to control permutations as in the 2-bit

and 3-bit cases. Each temporary variable is used in 4-bit chunks from

the low-order end. 8 cycles are necessary to use up each variable. A

cycle adds a 4-bit chunk into s0, then applies PERM1. Then the next

4-bit chunk is xored into s0, and PERM2 is applied.

Decryption is a straightforward reversal of encryption.

PERM1: 0x324f6a850d19e7cb /\* cycle notation (0B6D49851CF3E27)(A) \*/

PERM1I: 0xc3610a492b8dfe57 /\* inverse of PERM1 \*/

PERM2: 0x2b7e1568adf09c43 /\* cycles (0396D7A5F2CEB14)(8) \*/

PERM2I: 0x5c62e738d9a10fb4 /\* inverse of PERM2 \*/

The permutation defined by PERM1(N) is calculated by right shifting

PERM1 by N hex digits, and masking to 4 bits. So 0->b, 1->c, ...,

15->3. Note that "a" is a fixed point of PERM1, and thus of PERM1I,

in position 10. PERM1 is derived from pi by writing it in hex and

dropping duplicate hex digits. PERM1I is the inverse of PERM1, and is

used for decryption. PERM2 is used in the same way as PERM1. It is

derived from e (2.718).

As with blocksizes 2 and 3, an attacker can learn the parity of the

low-order bits of the sixteen bytes in the tmp array, if he learns the

entire map of 16 plaintext-ciphertext pairs.

Blocksizes 5 and 6 are handled similarly to blocksize 4. A longer tmp

array is used, to provide more pseudo-random material. This allows

all possible permutations of the 32 or 64 values to occur. For size

5, 3 words (192 bits) from the KX array are used. For size 6, 6 words

(384 bits) are used. In both cases, the starting word is

KX[16+2\*blocksize]. HPC-Long is called to encrypt the block of KX

words, using KX as the key expansion array, and using the same spice

as the call to HPC-Tiny. The output of the encryption is placed in a

temporary array. Each word of the output, starting with tmp[0], is

used to control a permutation engine operating on s0.

For blocksize 5, the engine inner loop runs 7 times. After each

cycle, the engine control variable T (from the tmp array) is right

shifted 9 bits. Since the shift is less than 10 bits, a few bits of T

are used in two cycles. The cycle code is

s0 ^= T

the low 4 bits of s0 are mapped with PERM1

s0 ^= s0 >>3

s0 += T>>blocksize

the low 4 bits of s0 are mapped with PERM2

The same scheme is used for blocksize 6. The inner loop runs only 6

times, and at the end of the cycle, T is right shifted 11 bits.

Blocksizes 7 to 16 use a somewhat different scheme. The same method

as for blocksizes 16-35 is used to generate a 10-word pseudo-random

array, tmp. (This includes a call to HPC-Long for a 512-bit

encryption, and an ad hoc routine to calculate tmp[8] and tmp[9].)

The 10 words are fed to the permutation engine, starting with T =

tmp[0].

Each cycle of the engine uses the lower order 2\*blocksize bits of T.

These are right shifted away at the end of each cycle, until all 64

bits of T are consumed. The cycle code is

s0 += T

s0 ^= KX [16\*I + (s0&15)] <<4

I is the index into the tmp array, going from 0...9

The low-order 4 bits of s0 are used in the KX fetch,

so they mustn't be changed. Thus the "<<4".

s0 is rotated right 4 bits

s0 ^= s0 >> LBH

s0 ^= T >> blocksize

s0 += s0 << (LBH+2)

s0 ^= Perma[s0&15]

s0 += s0 << LBH

LBH is (blocksize+1)/2, rounded down.

The array Perma is similar to the array Permb described under

HPC-Short. It is derived from pi instead of e, and is set up to allow

entries to be xored into s0, rather than added or subtracted. Because

Perma and Permai are only used in encryptions of <=15 bits, only the

low order 15 bits of the entries matter.

Perma[16]:

0x243F6A8885A308D3 ^0, 0x13198A2E03707344 ^1,

0xA4093822299F31D0 ^2, 0x082EFA98EC4E6C89 ^3,

0x452821E638D01377 ^4, 0xBE5466CF34E90C6C ^5,

0xC0AC29B7C97C50DD ^6, 0x9216D5D98979FB1B ^7,

0xB8E1AFED6A267E96 ^8, 0xA458FEA3F4933D7E ^9,

0x0D95748F728EB658 ^10, 0x7B54A41DC25A59B5 ^11,

0xCA417918B8DB38EF ^12, 0xB3EE1411636FBC2A ^13,

0x61D809CCFB21A991 ^14, 0x487CAC605DEC8032 ^15.

Permai[16]:

0xA4093822299F31D0 ^2, 0x61D809CCFB21A991 ^14,

0x487CAC605DEC8032 ^15, 0x243F6A8885A308D3 ^0,

0x13198A2E03707344 ^1, 0x7B54A41DC25A59B5 ^11,

0xB8E1AFED6A267E96 ^8, 0x452821E638D01377 ^4,

0x0D95748F728EB658 ^10, 0x082EFA98EC4E6C89 ^3,

0xB3EE1411636FBC2A ^13, 0x9216D5D98979FB1B ^7,

0xBE5466CF34E90C6C ^5, 0xC0AC29B7C97C50DD ^6,

0xA458FEA3F4933D7E ^9, 0xCA417918B8DB38EF ^12.

Decryption is done by reversing the encryption steps.

The inverse for s0 ^= Perma[s0&15] is s0 ^= Permai[s0&15].

For blocksizes 16-35, a more complicated approach is used. The first

part, setting up the tmp[] array, is the same as for blocksizes 7-15.

A 10-word temporary array is created. The first 8 words are the xor

of the spice and a block of 8 words from the KX array.

tmp[0] = spice[0] ^ KX[4\*blocksize+16]

etc.

tmp[7] = spice[7] ^ KX[4\*blocksize+23]

Then HPC-Long is called to encrypt the 512-bit quantity tmp[0...7].

The KX array is used as the key, and an ALL ZERO spice is used. The

encryption is in-place. After the encryption, tmp[8] and tmp[9] are

calculated as a kind of checksum. Each is initialized to tmp[7].

tmp[8] is marched through the iteration

tmp[8] += ((tmp[8]<<21) + (tmp[8]>>13)) ^ (tmp[i] + KX[16+i])

as i goes from 0...7. tmp[9] is the xor of the values of tmp[8],

including the initial value (tmp[7]) and after each cycle of the

iteration. (This is the end of the match with blocksizes 7-15.)

The 10-word tmp array is then processed in a way similar to the

shorter blocksizes, running a permutation engine on s0. Each word,

starting from tmp[0] and going through tmp[9], is processed in a loop.

Using T as the word from the tmp array, the inner cycle is

s0 += t

s0 ^= KX[s0&255] << 8

s0 &= LMASK

s0 is rotated right 8 bits

t >>= blocksize

The cycle is run until all 64 bits of T are processed. This will be 2

cycles for blocksizes 35-32, three for blocksizes 31-22, and four for

blocksizes 21-16. Then the next word of the tmp array is fetched, and

the inner cycle repeated.

Decryption reverses the encryption steps.

Encrypting Numbers and Dates

Because Hasty Pudding can encrypt any blocksize without data

expansion, it is feasible to use it for another objective: An

encrypted permutation of N objects, where N is not a power of 2.

For example, you might wish to encrypt a date as a date. Or you might

want to generate a random permutation, perhaps for playing a game of

cards. The idea can be extended to encrypting sets within their own

domain. For example, one might encrypt the printable ascii

characters; or the alphabet.

To encrypt a range of numbers from 0 to N-1: Let 2^K be the smallest

power of 2 that is >= N. If we are interested in encrypting dates, N

would be 365 (or 366 for a leap year). 2^K would be 512. The

encryption method is simple: set the plaintext equal to the number

to encrypt, and the blocksize to K. Call the encryption repeatedly,

stopping when the ciphertext is in the valid range. The expected

average number of calls is roughly 2^K/N, which is always < 2.

Decryption just runs the idea backward.

To encrypt a deck of cards, imagine the cards as numbers from 0 to 51.

Use a 6-bit blocksize. The permutation can be read as telling what

the Nth card in the shuffled pack is.

The idea extends easily to mapping printable ascii characters. Assume

there are 95 such characters, including "space". A 7-bit blocksize

will work nicely, mapping characters with values in the range

[32,126].

The ultimate extension of this idea is to replace the range test with

a membership predicate, which decides if a given bit string is a

member of the set-to-be-encrypted. The scheme will become impractical

if the targets are too thin, however.

We might try to encrypt 10-bit primes. Since there are only 172 such

primes, the average number of encryption steps will be about 6.

Because of its probabilistic nature, the scheme is unsuited for hard

real-time use, since the number of encryption steps required could

occasionally be very large.