Wheesht: an AEAD stream cipher.

Peter Maxwell (designer and submitter)
peter@allicient.co.uk

14th March 2014: v0.3

Abstract

Wheesht is an authenticated stream cipher with associated data. Internally, it uses the ARX paradigm and is heavily orientated towards architectures with 64-bit words. The message authentication follows the universal hashing style. The blocksize is 256-bits with an internal 512-bit state that is split in half to provide 256-bits of key material for encryption and the same for the authentication of each block. Wheesht allows blocks to be computed out-of-order. The keysize is 512-bits: 256-bits for the encryption and setting up per-block authentication calculations, and 256-bits for the final authentication calculation.

Performance on target platforms is at least comparable to similar ciphers and message authentication codes with the reference implementation performing encryption with authentication at around 7cpb for messages of 576 bytes in length.

1 Introduction

Wheesht is designed specifically for processors that can handle 64-bit words. Most consumer and server devices either fall into this category already or will do in the near future. Similarly, it offers the potential for servers to off-load cryptographic work to a GPU processor. The cipher is designed in such a manner as to calculate the keystream and authenication for each block at the same time, with optimizations available for blocks that are not the first block.

The design borrows heavily from Bernstein's Salsa20 cipher with some additional inspiration taken from the compactness and efficiency of Aumasson and Bernstein's SipHash. The message authentication code follows on from the universal hashing method, see `http://cr.yp.to/mac/poly1305-20050329.pdf`, section 1, "Genealogy" for a full history. Rather than using multiplication in a finite field for authentication as akin to GCM or Poly1305, the authenication mechanism in Wheesht uses the same round function that is used for the main block calculations. The motivation for this is two-fold: avoiding the risk of naive implementations not being constant-time when the processor does not natively support multiplication in a finite field; secondly, it keeps the code size smaller and avoids unnecessary complexity.

Strictly speaking the cipher can encrypt up to 2^{128} blocks, in other words 2^{136} bytes, however it is probably not particularly wise to try and encrypt large amounts of data under one session, i.e. under the same [key:public message number:secret message number] combination.

2 Specification

2.1 Preliminaries

Wheesht operates on little-endian 64-bit words. For each 256-bit (4-word) block of plaintext, Wheesht requires a 512-bit (8 word) state to operate on.

The following are required inputs for the cipher:

- Key, k: a 512-bit key split into two halves, the cipher key k_c and a final authentication key k_f where the four words of k_c are specified as $k_{c0}, k_{c1}, k_{c2}, k_{c3}$ and similarly for k_f;
- Plain Text (encryption), $P[0..n-1]$, or Cipher Text (decryption), $C[0..n-1]$: n blocks, normally 256-bits long excepting the last block which may be shorter;
- Public Message Number: a 128-bit public message number, n_p with high-word n_{p1} and low-word n_{p0}, set to zero if not used;
- Secret Message Number: a 128-bit public message number, n_p with high-word n_{s1} and low-word n_{s0}, set to zero if not used;
- Round Count: the number of rounds, t_m of PartRound we use for the main computation steps, i.e. for PartRound-n $n = t_m$;

- Final Rount Count: the number of rounds of PartRound we use for the final transform in the main block and final auth calculation, t_f;
- Required Length of Authentication Tag in Bits: *ataglen*; and,
- Authentication Tag (decryption only), *tag*: a 256-bit tag used to verify message integrity.

The following are required inputs for each block:

- Plain Text (encryption), p or Cipher Text (decryption), c: the plaintext or ciphertext block, normally 256-bit long and containing 4 words referenced by p_0, p_1, p_2, p_3 and c_0, c_1, c_2, c_3;
- Block Number, b_c and b_{ad}: two 128-bit block counters, one for the ciphertext and one for associated data, with high-word b_{c1} and low-word b_{c0} and similarly b_{ad1} and b_{ad0};
- Len, l: a unsigned 64-bit integer representating the number of bits to encrypt in that block, usually equal to the value 256 but on the last block this may be smaller; and,
- Mode Bits, *mode*: a bitmask with flags determining certain characteristics for the current block. The precise bitmask for *mode* is defined later in this section.

Wheesht also makes use of 8 64-bit constants, $q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7$ which are defined later in this section.

2.2 Notation

When not otherwise stated, all numeric constants are little-endian 64-bit words.

The following arithmetic operations are used:

- Exclusive-OR: written as $\oplus$
- Unsigned addition modulo 2^{64}: written as $+$
- Bitwise left rotation: written as $<<$

The notation that is normally used for functions is somewhat abused here. Instead of the standard mathematical notation, we simply donate a function, say, $F(x, y, z)$, and assume that the results can be returned into the specified variables.

2.3 Naming

The implementation of Wheesht is specified using three parameters - the authtag length ($ataglen$), main round count (t_m) and final round count (t_f) as follows:

Wheesht-t_m-t_f-$ataglen$

For example, Wheesht-3-1-256 uses three main rounds, one finalisation round and creates authentication tags of 256-bits. The key size is fixed at 512-bits so doesn't need specified.

The main round count, t_m, should not be less than 3.

Four parameter sets are specified for the purposes of CAESAR:

Wheesht-3-1-128
Wheesht-3-1-256
Wheesht-3-3-256
Wheesht-5-7-256

2.4 PartRound

There is a PartRound operation, $\theta(s_0, s_1, s_2, s_3)$ defined on a 4-word block, s_0, s_1, s_2, s_3 with four rotation constants, $r_I, r_{II}, r_{III}, r_V$ as follows.

$$
\begin{aligned}
s_0 = (s_0 + s_1); s_1 = (s_1 << r_I); s_1 = (s_1 \oplus s_0); s_3 = (s_3 + s_1);\\
s_2 = (s_2 + s_3); s_3 = (s_3 << r_{II}); s_3 = (s_3 \oplus s_2); s_1 = (s_1 + s_3);\\
s_0 = (s_0 + s_1); s_1 = (s_1 << r_{III}); s_1 = (s_1 \oplus s_0); s_3 = (s_3 + s_1);\\
s_2 = (s_2 + s_3); s_3 = (s_3 << r_{IV}); s_3 = (s_3 \oplus s_2); s_1 = (s_1 + s_3);
\end{aligned}
$$

A multiple operation of PartRound-n is defined, for n successive operations donated θ^n.

2.5 FullRound

The FullRound operation, $\omega(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7; x_0, x_1, x_2, x_3)$ is defined on an 8-word block, $s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7$ with an additional 4-word input x_0, x_1, x_2, x_3.

First mix in the 4-word input into the state:

$$
\begin{aligned}
s_1 = s_1 \oplus x_0;\\
s_3 = s_3 \oplus x_1;\\
s_5 = s_5 \oplus x_2;\\
s_7 = s_7 \oplus x_3;
\end{aligned}
$$

Next apply the PartRound-n function to the the two 256-bit blocks separately with $n = t_m$:

$$\theta^{t_m}(s_0, s_1, s_2, s_3);$$
$$\theta^{t_m}(s_4, s_5, s_6, s_7);$$

Finally, two words from each of the halfs are interchanged:

$$s_0 \leftrightarrow s_4;$$
$$s_2 \leftrightarrow s_6;$$

2.6 Main Block Calculation

The MainBlock operation, $\phi(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7; k_{c0}, k_{c1}, k_{c2}, k_{c3}, n_{s0}, n_{s1}, n_{p1}, n_{p0}, b_0, b_1, len, mode)$ is defined on an 8-word block, $s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7$ and the inputs as specified in the first section.

First the key is copied into each half of the 8-word state and XORed with the 8 constants. This ensures that $s_0 \neq s_4, s_1 \neq s_5, s_2 \neq s_6, s_3 \neq s_7$.

$$s_0 = k_{c0} \oplus q_0;$$
$$s_1 = k_{c1} \oplus q_1;$$
$$s_2 = k_{c2} \oplus q_2;$$
$$s_3 = k_{c3} \oplus q_3;$$
$$s_4 = k_{c0} \oplus q_4;$$
$$s_5 = k_{c1} \oplus q_5;$$
$$s_6 = k_{c2} \oplus q_6;$$
$$s_7 = k_{c3} \oplus q_7;$$

The MainBlock then proceeds as follows:

$$\omega(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7; n_{s0}, n_{s1}, n_{p0}, n_{p1});$$
$$\omega(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7; b_0, b_1, len, mode);$$
$$\theta^{t_f}(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7;);$$

Note the last PartRound-n uses $n = t_f$ not the usual $n = t_m$.

There is a swap between the two halfs of the state:

$$s_0 \leftrightarrow s_4;$$
$$s_2 \leftrightarrow s_6;$$

Finally, the key is reapplied:

$$
\begin{aligned}
s_0 &= s_0 \oplus k_{c0};\\
s_1 &= s_1 \oplus k_{c1};\\
s_2 &= s_2 \oplus k_{c2};\\
s_3 &= s_3 \oplus k_{c3};\\
s_4 &= s_4 \oplus k_{c0};\\
s_5 &= s_5 \oplus k_{c1};\\
s_6 &= s_6 \oplus k_{c2};\\
s_7 &= s_7 \oplus k_{c3};
\end{aligned}
$$

2.7 AuthBlock

There are two authentication functions: a per-block calculation and finalisation calculation. This is the per-block calculation.

The AuthBlock operation $\sigma(c_0, c_1, c_2, c_3; k_{t0}, k_{t1}, k_{t2}, k_{t3}; a_0, a_1, a_2, a_3;)$ takes the ciphertext as input $c_0..c_3$ along with the per-block auth key, $k_{t0}..k_{t3}$ and returns the per-block authentication hash $a_0..a_3$.

A 4-word state is created:

$$
\begin{aligned}
s_0 &= k_{t0} \oplus c_0;\\
s_1 &= k_{t1} \oplus c_1;\\
s_2 &= k_{t2} \oplus c_2;\\
s_3 &= k_{t3} \oplus c_3;
\end{aligned}
$$

The state is then passed through PartRound-n, where $n = t_m$.

$$\theta^{t_m}(s_0, s_1, s_2, s_3);$$

Calculate the result:

$$
\begin{aligned}
a_0 &= s_0 \oplus k_{t0};\\
a_1 &= s_1 \oplus k_{t1};\\
a_2 &= s_2 \oplus k_{t2};\\
a_3 &= s_3 \oplus k_{t3};
\end{aligned}
$$

2.8 AuthFinal

The AuthFinal operation, $\tau(k_{f0}, k_{f1}, k_{f2}, k_{f3}; n_{s0}, n_{s1}, n_{p1}, n_{p0}, b_0, b_1, len, mode; a_{f0}, a_{f1}, a_{f2}, a_{f3})$ is the second part to the authentication calculation and returns a 4-word finalisation result.

Similar to MainBlock, a 8-word state is created:

$$
\begin{aligned}
s_0 &= k_{f0} \oplus q_0; \\
s_1 &= k_{f1} \oplus q_1; \\
s_2 &= k_{f2} \oplus q_2; \\
s_3 &= k_{f3} \oplus q_3; \\
s_4 &= k_{f0} \oplus q_4; \\
s_5 &= k_{f1} \oplus q_5; \\
s_6 &= k_{f2} \oplus q_6; \\
s_7 &= k_{f3} \oplus q_7;
\end{aligned}
$$

Next we proceed in a manner akin to MainBlock:

$$
\begin{aligned}
\omega(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7; n_{s0}, n_{s1}, n_{p0}, n_{p1}); \\
\omega(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7; b_0, b_1, len, mode); \\
\theta_{t_f}(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7);
\end{aligned}
$$

Note the last PartRound-n uses $n = t_f$ not the usual $n = t_m$.

The last swap is not necessary here.

Finally, calculate:

$$
\begin{aligned}
a_{f0} &= s_0 \oplus s_4 \oplus k_{f0}; \\
a_{f1} &= s_1 \oplus s_5 \oplus k_{f1}; \\
a_{f2} &= s_2 \oplus s_6 \oplus k_{f2}; \\
a_{f3} &= s_3 \oplus s_7 \oplus k_{f3};
\end{aligned}
$$

2.9 AEADEncrypt

For authenticated encryption, each 256-bit = 4-word block of the plaintext message is XORed with the first half - $s_0...s_3$ of the relevant MainBlock result. For message end blocks that are not 256-bits in length, the input is padded with zeros up to the blocksize and the *len* field is set to the number of bits of message in the last block. The ciphertext need only be as long as the original plaintext but the padded blocks are required for the authentication calculation. The second half of the MainBlock result is used to key the authentication calculation for this block. The result of the authentication calculation for each block is added word-wise to, $auth_0...auth_3$ (initialised to zero at the start of the session), creating a running sum; at the end of encrypting the plaintext and processing the associated data, a finalisation authentication calculation is done and that is added word-wise onto $auth_0...auth_3$, forming the authentication tag for the session.

At the start of each session, initiliase $auth_0, auth_1, auth_2, auth_3$ to zero and perform the encryption first before moving onto the associated data. The block counters for encryption and the associated data are separate as defined at the start of this section.

So for plaintext block $P[i]$, $i = b_{c1}.2^{64} + b_{c0}$ specified by words p_0, p_1, p_2, p_3, first calculate:

$$\phi(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7; k_{c0}, k_{c1}, k_{c2}, k_{c3}, n_{s0}, n_{s1}, n_{p1}, n_{p0}, b_{c0}, b_{c1}, len, mode)$$

Now, the ciphertext is calculated thus:

$$\begin{aligned} c_0 &= p_0 \oplus s_0; \\ c_1 &= p_1 \oplus s_1; \\ c_2 &= p_2 \oplus s_2; \\ c_3 &= p_3 \oplus s_3; \end{aligned}$$

The per-block authentication hash is calculated as:

$$\sigma(c_0, c_1, c_2, c_3; s_4, s_5, s_6, s_7; a_0, a_1, a_2, a_3;);$$

And is added into $auth_0 ... auth_3$:

$$\begin{aligned} auth_0 &= auth_0 + a_0; \\ auth_1 &= auth_1 + a_1; \\ auth_2 &= auth_2 + a_2; \\ auth_3 &= auth_3 + a_3; \end{aligned}$$

Once all the ciphertext is calculated, the associated data is parsed in the same manner by calculating for each block:

$$\phi(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7; k_{c0}, k_{c1}, k_{c2}, k_{c3}, n_{s0}, n_{s1}, n_{p1}, n_{p0}, b_{ad0}, b_{ad1}, len, mode)$$

Except this time the first half of the result of ϕ is discarded and the second half is used for the per-block authentication calculation as before:

$$\sigma^n(ad_0, ad_1, ad_2, ad_3; s_4, s_5, s_6, s_7; a_0, a_1, a_2, a_3;);$$

And again is added into $auth_0 ... auth_3$:

$$auth_0 = auth_0 + a_0;$$
$$auth_1 = auth_1 + a_1;$$
$$auth_2 = auth_2 + a_2;$$
$$auth_3 = auth_3 + a_3;$$

Finally, once the associated data is parsed, AuthFinal is used (note that the block counters are specified as the word-wise addition of the ciphertext and associated data counter, and that the key used here is k_f):

$$\tau(k_{f0}, k_{f1}, k_{f2}, k_{f3}; n_{s0}, n_{s1}, n_{p1}, n_{p0}, b_{c0} + b_{ad0}, b_{c1} + b_{ad1}, ataglen, mode; a_{f0}, a_{f1}, a_{f2}, a_{f3})$$

Note: the *len* field is specified using *ataglen*.

Now the authentication tag for the session is calculated:

$$auth_0 = auth_0 + a_0;$$
$$auth_1 = auth_1 + a_1;$$
$$auth_2 = auth_2 + a_2;$$
$$auth_3 = auth_3 + a_3;$$

The authentication tag is appended onto the end of the ciphertext stream.

2.10 AEADDecrypt

The decrypt operation proceeds in the obvious manner and needs not be explicitly defined here.

2.11 *mode* bits

The following constants are defined:

```
CRYPT_CIPHERTEXTBLOCK = 0x01
CRYPT_ADBLOCK = 0x02
CRYPT_AUTHFINAL = 0x04
CRYPT_HASNSEC = 0x08
CRYPT_HASNPUB = 0x10
CRYPT_LASTBLOCK = 0x20
```

The *mode* is set depending on context as follows:

- When processing ciphertext blocks, the `CRYPT_CIPHERTEXTBLOCK` bit is set. If `CRYPT_ADBLOCK` is set then `CRYPT_ADBLOCK` cannot be set.

- When processing associated data blocks, the `CRYPT_ADBLOCK` bit is set. If `CRYPT_ADBLOCK` is set then `CRYPT_CIPHERTEXTBLOCK` cannot be set.
- When processing the last block of ciphertext or associated data, the `CRYPT_LASTBLOCK` bit must be set.
- In this specifcation both `CRYPT_HASNSEC` and `CRYPT_HASNPUB` must always be set. A cut-down version of the algorithm may be specified in the future where the message numbers are not used but for now they must be set.
- For the final auth calculation, `CRYPT_AUTHFINAL` and `CRYPT_LASTBLOCK` are set but `CRYPT_CIPHERTEXTBLOCK` and `CRYPT_ADBLOCK` are not.

Examples follow:

- For a ciphertext block that is not the last block the *mode* is set to `CRYPT_CIPHERTEXTBLOCK || CRYPT_HASNSEC || CRYPT_HASNPUB` .
- For a ciphertext block that is the last block the *mode* is set to `CRYPT_CIPHERTEXTBLOCK || CRYPT_HASNSEC || CRYPT_HASNPUB || CRYPT_LASTBLOCK` .
- For an associated data block that is not the last block the *mode* is set to `CRYPT_ADBLOCK || CRYPT_HASNSEC || CRYPT_HASNPUB` .
- For an associated data block that is the last block the *mode* is set to `CRYPT_ADBLOCK || CRYPT_HASNSEC || CRYPT_HASNPUB || CRYPT_LASTBLOCK` .
- For the final auth calculation, *mode* is set to `CRYPT_AUTHFINAL || CRYPT_HASNSEC || CRYPT_HASNPUB || CRYPT_LASTBLOCK` .

2.12 *len* Values

For the sake of clarity, *len* should be set to the value `256` for all ciphertext and associated data blocks that are not the last block. When a ciphertext or associated data block is the last block, *len* should be the number of bits in that last block.

For the final auth calculation, *len* should be the length of the desired authentication tag, which is normally `256` but may be specified lower if a shorter tag is more appropriate.

2.13 Constants

The rotation constants are defined as:

r_I = 9
r_{II} = 29
r_{III} = 33
r_{IV} = 41

The 8 word constants $q_0..q_7$ are as follows:

q_0 = 0x5720796d6f6f6c47
q_1 = 0x2073277265746e69
q_2 = 0x27617761206f6f6e
q_3 = 0x742074666173203b
q_4 = 0x6c74736577206568
q_5 = 0x6565726220276e69
q_6 = 0x77616c622073657a
q_7 = 0x000a38303831202e

which is just the conversion of the the string *"Gloomy Winter's noo awa'; saft the westlin' breezes blaw. 1808\n"* into 8 little-endian words. The string itself is the first line of Robert Tannahill's lyrics for "Gloomy Winter's Noo Awa" along with the year it was written.

3 Security Goals

The security goals are specified as follows.

Requirement	Bits
"Confidentiality for the plaintext"	256
"Confidentiality for the secret message number"	128
"Integrity for the plaintext"	256
"Integrity for the associated data"	256
"Integrity for the secret message number"	128
"Integrity for the public message number"	128

Note: **the public and secret message numbers together cannot be reused and must be considered a nonce**. If a key, public message number and secret message number are used to encrypt two different messages then both confidentiality and integrity will be entirely lost.

Wheesht is designed such that any execution paths dependent on secret data execute in constant time. There are potential optimizations but they are dependent on public data.

4 Security Analysis

4.1 Diffusion

The PartRound function achieves full diffusion after 3 rounds. This ensures that every bit in the input parameters is guaranteed to affect the whole 256-bits of state that PartRound-3 operates on.

Each FullRound operation swaps two of the four words in each 4-word state to ensure the 8-word state is altered at every FullRound, and consequentially each input bit will affect the full 8-word state.

4.2 General Remarks

Using the same assumption as the security analysis for Salsa20 - specifically that the key is assumed to be a uniform random sequence of bytes, and there is no reuse of the public-secret message number pair - then it is likely that Wheesht produces output that is indistinguishable from from "perfect" ciphertexts.

4.3 Security of Authentication

The message authentication is based on the idea of "universal hashing" akin to Galois Counter Mode, Poly1305, etc. The temporary authentication key for each block which is obtained from the second half of the main block calculation serves as a means to create a universal hash. The final auth block is essentially calculated in a similar manner to the main block calculation but under a separate key, which is why the total key length required for Wheesht is 512-bits.

It is presumed to be infeasible to break the authentication as the attacker cannot predict the temporary auth key for each block, and the temporary auth key is dependent on all the input parameters (including the block counter). The attacker similarly cannot predict the final auth block as they have no knowledge of the key and none of those key bits were used during the rest of the calculation.

5 Features

Wheesht offers a number of useful features:

- Reusing a simple round function: the basic round function is used both for encryption and authentication, obviating the requirement for difficult calculations in prime fields.
- Constant time: every operation that depends on secret data is constant time, reducing the window for side-channel attacks.
- ARX construction: the cipher is based on the well tested Addition-Rotate-XOR paradigm.
- Simple: the design is simple and robust. Despite the somewhat protracted definition herein, the actual cipher is very simple.
- Light-weight: Wheesht can be implemented with small code or hardware cost.
- Speed: the performance is at least comparable with similar ciphers.
- Out-of-order calculation: blocks can be calculated out-of-order or in parallel, this offers many advantages in terms of performance enhancements and flexibility.
- Optimized register use: the core round function uses 8 registers, which is a deliberate attempt to reduce the number of memory transfers required on the critical path.

- Advantage over AES-GCM: the design of Wheesht does not require SBox lookups nor does it require multiplication in finite fields. Wheesht is also much simpler than AES-GCM, which reduces potential code-size and the chance for errors during implementation. Without native AESNI and finite field arithmetic support on the CPU, Wheesht is likely to be faster in an optimized implementation.

6 Design Rationale

The designer/designers have not hidden any weaknesses in this cipher.

A weakness would be difficult to conceal. The most likely problem area is the number of final transformation rounds t_f, which the designer has yet to perform a full analysis on before recommending a fixed value. The constants $q_0 ... q_7$ could be specified such that the hamming weight of $q_0 \oplus q_4, q_1 \oplus q_5, q_2 \oplus q_6, q_3 \oplus q_7$ is small; the actual constants were chosen using a line from a favourite traditional song but can be changed by the CAESAR panel. The rotation constants will affect how fast diffusion is obtained, however there is a large number of combinations that provide the same diffusion performance and the CAESAR panel are welcome to reselect the rotation constants. Otherwise, the design has been kept simple so that it is not feasbile that any weakness could be deliberately hidden.

There were several objectives that motivated the design.

- Ensuring the encryption and authentication were in a sense unified and to avoid using disparate methods for each.
- Target 64-bit platforms as there is a large proportion of processors with that word length.
- Design the algorithm in a manner such that the core round function only required eight registers on the CPU, to minimize memory accesses.
- To allow out-of-order computation of blocks, which adds flexibility and parallization options.
- Avoid potentially troublesome operations, for example avoiding multiplation in finite fields.
- Ensure that all operations on secret data are constant-time.

By calculating on a 512-bit = 8-word block with a round function that operates independently on each half, a certain orthogonality can be obtained (each half starts with a copy of the 256-bit key k_c but XORed with different constants). For example, it is difficult to exploit a related-key attack or otherwise alter the inputs to the attacker's advantage because by doing so on one half ensures the effect on the other half is undesirable.

The second advantage to this structure is that the 8-word state can be split to provide one half for the keystream and the other for authentication. The block

cannot be inverted with only half the data and the authentication hash for that block cannot be predicted from the keystream block either.

The input parameters are partially layered into the state between the round function to reduce the degrees of freedom an attacker in control of some inputs may have.

The round function itself was arrived at after experimentation: essentially, it offers fast diffusion properties. The rotation constants were arrived at by first optimizing for diffusion - running the round function with OR instead of ADD and XOR, with a single 1 bit in the input - and then from the resulting candidates selecting constants so the difference from 8 in each is minimized (a concession for older processors that can only perform rotations one bit at a time).

The order in which the input parameters for each block are mixed-in was not determined by accident. It allows a pre-calculation to be preformed in the first block which can be used for later blocks, i.e. the public and secret numbers are done first as they do not change for the session so further blocks can proceed from this saved state.

7 Intellectual Property

As far as I know, Wheesht does not infringe any intellectual property rights.

If any of this information changes, the submitter/submitters will promptly (and within at most one month) announce these changes on the crypto-competitions mailing list.

8 Consent

The submitter/submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitter/submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitter/submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitter/submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitter/submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitter/submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.