Default Hash Function

The default hash function is defined as the following C# function:

```
static byte[] Hash(byte[] msg)
{
    UInt128 state = 1;
    byte[] block;
    int i;

    // Create hash value (compression function)
    for (i = 0; i < msg.Length; i++) state += (state << 8) + (msg[i] + 1);
    block = BitConverterEx.GetBytes(state);
    if (BitConverter.IsLittleEndian) Array.Reverse(block);

    // Diffuse hash value (kind of)
    for (i = block.Length-2; i >= 0; i--) block[i] ^= block[i + 1];

    return block;
}
```

The internal state s_i of the compression function is defined with its previous value s_{i-1} and the current message byte M_i as follows:

$$s_i = \left(\left(\underbrace{257}_{\text{prime number } p} \cdot s_{i-1}\right) + \left(\underbrace{M_i + 1}_{1 \ldots 256 < p}\right)\right) \bmod 2^{128}$$

After the hash value creation, the most volatile byte is the last byte of the hash value (assuming big Endian computation). This byte is successively exclusive ORed with the previous lesser volatile bytes, yielding the first byte as being the new most volatile byte.
The compression function starts to discard state bits after 15 message bytes have been processed. It is therefore at least collision resistant for a message length of up to 120 bits.
An alternative hash function is a PRNG (= pseudorandom number generator) which is seeded with the state after the compression function of the default hash function. The final output value is exclusive ORed with the seed to address a reversible PRNG.
An alternative keyed hash function is a 128-bit block permutation with the state after the compression function of the default hash function as input message. The output message is exlusive ORed with the 128-bit number of message bytes of the default hash function in little Endian format, followed by a further 128-bit block permutation with the same key. Note that the block of the compression function is in case of a keyed hash reversed if big Endian computing instead of little Endian computing is used (for performance reasons).

Default Block Permutation

The default block permutation is defined as the following C# function:

```
static void ObfuscateBlock(byte[] key, byte[] block)
{
    if (key.Length > 7) throw new ArgumentException();
}
```

Note that only the forward transformation of a block permutation is used by the bootloader and not the backward transformation. Hence, a block deobfuscating method is not required.

General Message Authentication Code

A number message *num*(*i*), which can be represented with the bytes i_3, i_2, i_1, and i_0 with $i = i_3 \cdot 256^3 + i_2 \cdot 256^2 + i_1 \cdot 256 + i_0$, is defined as the byte sequence (i_0) for $i < 255$, or (255, i_0, i_1) for $i < 65535$, or (255, 255, 255, i_0, i_1, i_2, i_3) otherwise.

A length extension method is defined as follows:

length(*M*) = length in bytes of message *M*

len(*M*) = *num*(*length*(*M*)) || *M*

The helper function $MACS_K(s, \ldots)$ is defined as follows:

$MACS_K(s, \ldots) = hash(\ 0 \ ||\ hash(\ num(s) \ ||\ len(K) \ ||\ len(N) \ ||\ num(t) \ ||\ len(M)\ )\ ),$
for $length(N) > 0$

or

$MACS_K(s, \ldots) = hash(\ 0 \ ||\ hash(\ num(s) \ ||\ len(K) \ ||\ 0 \ ||\ num(t) \ ||\ num(d)\ )\ ),$
for $length(N) = 0$

with

hash function *hash*(*M*) of message *M*,
zero byte 0,
segment number *s* starting with 1,
key *K*,
nonce *N*,
type *t* (or *t* = message number *i*),
message *M*,
device number *d*, or *d* = 0 for a missing device number

The helper function $MACS_K(s, \ldots)$ can optionally be reduced to the inner hash function if the hash function is *Keccak*:

$MACS_K(s, \ldots) = Keccak(\ num(s) \ ||\ len(K) \ ||\ len(N) \ ||\ num(t) \ ||\ len(M)\ )$
for $length(N) > 0$

or

$MACS_K(s, \ldots) = Keccak(\ num(s) \ ||\ len(K) \ ||\ 0 \ ||\ num(t) \ ||\ num(d)\ )$
for $length(N) = 0$

The general *MAC* (= message authentication code) is defined as follows:

$MAC_K(args) = truncate_l(\ MACS_K(1, args) \ ||\ MACS_K(2, args) \ ||\ MACS_K(3, args) \ ||\ \ldots\ )$

with

list of arguments *args*

$truncate_l(M)$ = truncate the message *M* after *l* bits, or after 128 bits if *l* is zero

Missing function arguments are taken either as byte sequences of size 0, or as the value 0. Double hashing *hash*(*hash*(*M*)) is also used by [1] and by [9], [10]. The concatenated arguments from $MACS_K$ are defined in a way that they can all be parsed back into separate function arguments (see also [1]). The first zero byte is used to prevent identical arguments for the hash function. If a keyed hash with $Permutation128_K$ is used, the message part $len(K)$ of $MACS_K$ is replaced with $len(Permutation128_K(0))$ in order not to reuse the key for different algorithms directly:

$MACS_K(s, \ldots) = hash_K(\ 0 \ ||\ hash_K(\ num(s) \ ||\ len(Permutation128_K(0)) \ ||\ len(N) \ ||\ num(t) \ ||$
$len(M)\ )\ ),$
for $length(N) > 0$

or

$MACS_K(s, \ldots) = hash_K(\ 0 \ ||\ hash_K(\ num(s) \ ||\ len(Permutation128_K(0) \ ||\ 0 \ ||\ num(t) \ ||\ num(d)\ )\ ),$
for $length(N) = 0$

Key Derivation

A master key K is used to derive all the other keys:

Obfuscation key $K_o = MAC_K(t=0, l=56)$

Authentication key $K_a = MAC_K(t=1)$

Nonce key $K_n = MAC_K(t=2, l=56)$

Obfuscated Nonce

The 8 bytes of the deobfuscated nonce $\hat{N} = (\hat{N}_0 \hat{N}_1 \hat{N}_2 \hat{N}_3 \hat{N}_4 \hat{N}_5 \hat{N}_6 \hat{N}_7)$ are defined as follows:

$\hat{N}_0$ = 255 · Milliseconds part of the UTC timestamp / 999

$(\hat{N}_1 \hat{N}_2 \hat{N}_3 \hat{N}_4)$ = Seconds since January 1, 1970 of the UTC timestamp in little Endian format (representable until February 6, 2106)

$(\hat{N}_5 \hat{N}_6)$ = User defined 16-bit index in little Endian format

$\hat{N}_7$ = 0 (Reserved)

The obfuscated nonce *Nonce* is defined with the 64-bit block permutation *Permutation64* as follows:

$$Nonce = Permutation64_{Kn}(\hat{N})$$

Authenticated Message Digest

The obfuscated message (→ encrypt-then-authenticate, see also [1]) and all other necessary items to deobfuscate the message are passed into the general *MAC* (→ Horton Principle, see also [1]):

$T_i = MAC_{Ka}(N=Nonce, M=C_i, t=i)$

Cheating (MAC as the 128-bit Block Permutation)

To save programming memory, the 128-bit block permutation *Permutation128* is allowed to be implemented with a hash function (kind of cheating). Though, block doublets for different plaintext blocks may occur. This is also the reason why a hash function cannot be used as the 64-bit block permutation which is used only for nonce obfuscation. However, these block doublets are less troublesome than the default 128-bit block permutation. For performance reasons, the output value of *Permutation128* is allowed to be of the same size as the output value of the hash function.

$Permutation128_{Ko}(P) = MAC_{Ko}(N=P, l=8 \cdot length(hash))$

Alternatively, as long as a more secure hash function than the default hash function is used, *Permutation128* can also be defined as $Permutation128_{Ko}(M) = hash(K_o \,||\, M)$. Note that this construction is prone to length extension attacks and partial message collision attacks (see also [1]), but the hash function is not used for authentication.

Cheating (Firmware Obfuscation with MAC)

To save even more programming memory, the following obfuscation algorithm can be defined:

$C_i = MAC_{Ko}(N=Nonce, t=i, l=8 \cdot length(M_i)) \oplus M_i$

with a 64-bit *Nonce*.

Primitives

I am not a cryptographer, and thus, the following text should be taken with a grain of salt. The default hash function, the default 128-bit block permutation and the default 64-bit block permutation of the firmware obfuscation are insecure. They do neither guarantee encryption nor authentication.

Xorshift128+ is a fast non-cryptographically secure PRNG.

Keccak and *Whirlpool* are cryptographically secure hash functions.

Camellia and *Rijndael* are cryptographically secure 128-bit block permutations.

IDEA is a cryptographically secure 64-bit block permutation.

To my knowledge, *Xorshift128+*, *Keccak*, *Whirlpool* (ISO/IEC 10118-3:2004), *Camellia* (ISO/IEC 18033-3:2010), *Rijndael* (ISO/IEC 18033-3:2010) and *IDEA* are neither of US origin nor of Canadian origin. Therefore, they can be used to protect intellectual property without necessarily infringing the US Export Control (Note that this also presumes that the implementation of these algorithms is neither of US origin nor of Canadian origin).

Neither HMAC, CMAC, CBC-MAC, CCM Mode, EAX Mode nor GCM Mode

HMAC, *CMAC*, *CBC-MAC* or authenticated obfuscation with either *CCM*, *EAX* or *GCM* mode is of US origin.

No Hybrid Obfuscation/Encryption

One reason for a symmetric obfuscation/encryption, instead of a hybrid one, is that there is no benefit if using the latter. Because as soon as the deobfuscating/decryption key is unveiled, the asymmetric obfuscation/encryption can be read as plaintext, and thus, making a hybrid obfuscation/encryption needless: An adversary can just reuse the readable symmetric keys and replicate the unmodified asymmetrically obfuscated/encrypted part. Another reason is that asymmetric obfuscation/encryption is usually of US origin.

No Destruction of the Keys

The keys are never destructed: The generation of the obufscated firmware files is assumed to be done on a secure computer. The bootloader can also initialize/clear its RAM before calling the application which is, by the way, not necessarily required, because the keys are also readable from the microcontroller‘s ROM.

Bibliography

[1] Niels Ferguson, Bruce Schneier, Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010. ISBN: 978-0-470-47424-2.

[2] Joan Daemen, Vincent Rijmen. *The Design of Rijndael*. Springer-Verlag, 2002. ISBN: 3-540-42580-2.

[3] Tom St Denis, Simon Johnson. *Kryptographie für Entwickler*. Franzis Verlag GmbH, 2017. ISBN: 978-3-645-60543-4.

[4] Sebastiano Vigna. *Xorshift128+*. http://xoroshiro.di.unimi.it/xorshift128plus.c (Feb. 9, 2019)

[5] Paulo S. L. M. Barreto. *The Whirlpool Hash Function*.
https://web.archive.org/web/20171129084214/http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html (Feb. 12, 2019)

[6] Marco Quinten. *C# implementation of Whirlpool*.
https://gist.github.com/SplittyDev/43ba394c18c4b86fef9b (Feb. 12, 2019)

[7] Vincent Rijmen. *The Rjindael Page*.
https://web.archive.org/web/20051228003819/http://www.iaik.tu-graz.ac.at/research/krypto/AES/old/%7Erijmen/rijndael/ (Feb. 12, 2019)

[8] *Bouncy Castle* (C# IDEA implementation).
http://www.bouncycastle.org/csharp/ (Jun. 19, 2019)

[9] *The Bitcoin Protocol*.
https://en.bitcoin.it/wiki/Protocol_documentation (Feb. 17, 2019)

[10] *Hashcash*.
https://en.bitcoin.it/wiki/Hashcash#Double_Hash (Feb. 20, 2019)

[11] *Tagged Stream Format*.
https://www.tellert.de/?product=tsf (Jun. 17, 2019)

[12] Chris Doty-Humphrey. *Practically Random*.
https://sourceforge.net/projects/pracrand/ (Feb. 12, 2019)

[13] Don Ho. *Notepad++*.
https://notepad-plus-plus.org/ (Feb. 13, 2019)

[14] Maël Hörz. *HxD*.
https://mh-nexus.de/de/programs.php (Feb. 13, 2019)

[15] *TTY Emulator.*
https://web.archive.org/web/20110717111702/http://www.ttyemulator.com/InstallerFiles/TTYEmulPEInstaller.exe (Feb. 13, 2019)