**Just to note, this is work in progress (unfinished)**

**Summary of Asmcodes series**

**Introduction**

Between November 2015 and March 2017, a number of cryptographic algorithms specifically designed for software implementation were selected for optimization in C and x86 assembly code.

The algorithms consisted of block and stream ciphers, cryptographic hash functions and some modular arithmetic functions necessary for asymmetric key exchange and verification of digital signatures.

The purpose of this work was to evaluate the suitability of such algorithms for resource constrained environments which could be anything from a small microcomputer to a block of memory.

Although the vast majority of low resource devices do not use the x86 architecture, I think the results presented here could offer valuable insight for anyone searching for cryptographic algorithms suitable for resource constrained environments. Most of the assembly codes were a joint effort with Peter Ferrie. [1]

**Desirable properties**

The criterion for selection was very simple:

* Patent free
* Moderate to high level of security
* Small space required in for both ROM/RAM with RAM being a lesser consideration.
* Simple implementation

Many ciphers despite their popularity don’t meet the criteria. Some ciphers are strong but complicated to implement or have large resource requirements.

Blowfish for example while considered immune to many cryptographic attacks uses large lookup tables to create key dependent s-boxes which cannot be calculated at runtime. This requires both large amounts of RAM and ROM therefore making it unsuitable for low resource devices. The same is true for ciphers such as MARS, Camellia, CAST and cryptographic hash algorithms like Streebog or Whirlpool.

The stream cipher HC-256 although incredibly small in ROM takes as much RAM as Blowfish since it uses a similar function for generating key stream. The assembly implementation for example uses 20KB of RAM for key generation and 8KB lookup tables.

For the windows operating system, allocating 20KB requires a small amount of code to perform a *page probe* in order to prevent exceptions occurring.

MARS also uses large lookup tables and again can’t be calculated at runtime. In the following table is a list of block and stream ciphers that were examined for implementation. The algorithm rows highlighted in grey were implemented but due to patents will not be made available in final library. Those in red are considered obsolete.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Type** | **Implemented** | **Low Resource** |
| **AES-256** | Block | Yes | Yes |
| **Serpent-256** | Block | Yes | Yes |
| **Twofish-256** | Block | Yes | Yes |
| **CAST-256** | Block | No |  |
| **ThreeFish-256** | Block | Yes | Yes |
| **Noekeon** | Block | Yes | Yes |
| **Speck** | Block | Yes | Yes |
| **Camellia-256** | Block | No |  |
| **MARS** | Block | No |  |
| **Kalyna** | Block | No | U |
| **Bel-T-256** | Block | Yes | Yes |
| **Kuznyechik** | Block | Yes | Yes |
| **SM4** | Block | Yes | Yes |
| **Blowfish** | Block | Yes | No |
| **DES** | Block | Yes | No |
| **RC5** | Block | Yes | Yes |
| **RC6** | Block | Yes | Yes |
| **ChaCha20** | Stream | Yes | Yes |
| **HC-256** | Stream | Yes | Yes/No |
| **Salsa20** | Stream | Yes | Yes |
| **Rabbit** | Stream | Yes | Yes |
| **SM3** | Hash | Yes | Yes |
| **SHA3** | Hash | Yes | Yes |
| **BLAKE2** | Hash | Yes | Yes |
| **SHA2** | Hash | Yes | Yes |
| **Half Sip Hash** | Hash | Yes | Yes |
| **Chaskey** | MAC | Yes | Yes |
| **Poly1305** | MAC | Yes | Yes |
| **CubeMAC** | MAC | Yes | Yes |
| **CubeHash** | Hash | Yes | Yes |
| **MD4** | Hash | Yes | Yes |
| **MD5** | Hash | Yes | No |
| **SHA1** | Hash | Yes | Yes |

Based on the results, AES-256 is favourable as block cipher with Salsa20 as stream cipher and SHA3 as hash. The security of AES-256 is constantly being examined since it was made a standard in 2000 by NIST. Salsa20 was examined extensively as part of the eSTREAM competition and although similar to ChaCha20 by the same author, the latter has probably not been scrutinized as much as Salsa (open to correction).

**Symmetric Encryption**

The table below shows the differences between ciphers in byte size for both compiler generated and hand written x86 assembly. X64 assembly would undoubtedly result in larger code but are not implemented. Percentages shown here are approximations of hand written assembly versus assembly generated by compiler.

|  |  |  |  |
| --- | --- | --- | --- |
| **Cipher** | **Type of cipher** | **ROM (x86 ASM)**  **Bytes** | **ROM (MSVC)**  **Bytes** |
| **DES** | Block | 1038 | 1381 |
| **TWOFISH-256** | Block | 610 | 1242 |
| **SERPENT-256** | Block | 530 | 1027 |
| **KUZNYECHIK** | Block | 615 | 928 |
| **SM4** | Block | 502 | 690 |
| **AES-256** | Block | 377 | 827 |
| **THREEFISH-256** | Block | 371 | 747 |
| **RABBIT** | Stream | 457 | 700 |
| **BEL-T** | Block | 490 | 656 |
| **NOEKEON** | Block | 292 | 431 |
| **SALSA20** | Stream | 245 | 436 |
| **CHACHA20** | Stream | 241 | 424 |
| **RC6** | Block | 247 | 408 |
| **HC-256** | Stream | 272 | \*354 |
| **RC5** | Block | 167 | 237 |
| **SPECK** | Block | 105 | 139 |

\*= requires additional hidden code to increase stack size

**Cryptographic Hash, MAC and checksum algorithms**

The main algorithms selected here were from the SHA-3 competition although HMAC-SHA-2 was also included. MD4, MD5 and SHA-1 were merely included for comparison but were not seriously considered for use in the end.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Type** | **ROM (x86 ASM)**  **Bytes** | **ROM (MSVC)**  **Bytes** |
| **SHA2-256** | Hash | 653 | 1324 |
| **MD5** | Hash | 685 | 1202 |
| **BLAKE2-256** | Hash | 509 | 1027 |
| **SHA3** | Hash | 460 | 879 |
| **MD4** | Hash | 396 | 535 |
| **SHA-1** | Hash | 447 | 500 |
| **Poly1305** | MAC | 332 | 507 |
| **Chaskey** | MAC | 234 | 346 |
| **Cube Hash** | Hash | 201 | 448 |
| **CubeMAC128** | MAC | 196 | 380 |
| **Half Sip Hash 32** | PRF | 142 | 261 |
| **CRC32** | Checksum | 50 |  |
| **CRC32-C** | Checksum | 46 |  |

Based on implementations, SHA-3 has advantage for being the standard and will undoubtedly receive more scrutiny than other algorithms listed. Cube Hash is incredibly compact but also slow unless using a vectorized version. The nice thing about Cube Hash is the ability to tweak parameters and would be useful for very small applications.

**Key Exchange**

Due to the simplicity of factoring based public key exchange, only RSA and Diffie-Hellman-Merkle were examined. Elliptic Curve and Lattice based methods result in much more code although could be considered as an alternative if space isn’t important.

The main function used for both methods of key exchange is Modular Exponentiation.

Obviously using RSA is much faster but I can’t tell you which is more secure.

**Final Results**

Three configurations could be considered.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Algorithm** | | **Size (MSVC)** | **Size (asm)** |
| **Symmetric** | AES-256 | 760 | | 377 |
| **Key Exchange** | RSA-2048 | 844 | | 138 |
| **MAC** | SHA3 | 879 | | 460 |
| **Total** |  | **2483** | | **975** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Algorithm** | | **Size (MSVC)** | **Size (asm)** |
| **Symmetric** | Salsa20 | 436 | | 245 |
| **Key Exchange** | RSA-2048 | 867 | | 138 |
| **MAC** | Cube Hash | 448 | | 201 |
| **Total** |  | **1751** | | **584** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Algorithm** | | **Size (MSVC)** | **Size (asm)** |
| **Symmetric** | ChaCha20 | 424 | | 241 |
| **Key Exchange** | RSA-2048 | 844 | | 138 |
| **MAC** | CubeMAC | 380 | | 196 |
| **Total** |  | **1775** | | **575** |

**Future Work**

The CAESAR competition is expected to be finalized by December 2017 at which time this document will hopefully be updated to reflect a number of algorithms chosen.

**Summary**

The numbers presented in all tables are approximations since further modifications would decrease the sizes significantly. For example, if using CTR mode for symmetric encryption, the decryption functionality can be removed for some algorithms which include inverse tables.

The parameters of MAC can be fixed. Multiple calls can be reduced to one. Multiple functions can be “glued” together. Many versions could shave hundreds of bytes from at least the assembly code level.

Although it’s entirely possible to write offset independent code using pure C and such code is easily ported to other architectures, the compiler could never possibly compete with experienced assembly programmers given the current compilers available.

Depending on the design of the algorithms, a compiler does come close to hand-written assembly but is usually not capable of reducing as much as assembly programmer.

**References**

1. Homepage of Peter Ferrie

<http://pferrie.host22.com/>

1. AES: the Advanced Encryption Standard

<https://competitions.cr.yp.to/aes.html>

1. eSTREAM: the ECRYPT Stream Cipher Project

<https://competitions.cr.yp.to/estream.html>

1. SHA-3: a Secure Hash Algorithm

<https://competitions.cr.yp.to/sha3.html>

1. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness

<https://competitions.cr.yp.to/caesar.html>