ULCL
An Ultra-lightweight Cryptographic Library for Embedded Systems

George Hatzivasilis[1], Apostolos Theodoridis[2], Elias Gasparis[2] and Charalampos Manifavas[3]
[1] *Dept. of Electronic & Computer Engineering, Technical University of Crete, Akrotiri Campus, 73100 Chania, Crete, Greece*
[2] *Dept. of Computer Science, University of Crete, Voutes Campus, 70013 Heraklion, Crete, Greece*
[3] *Dept. of Informatics Engineering, Technological Educational Institute of Crete, Estavromenos, 71500 Heraklion, Crete, Greece*

Keywords: Cryptographic Library, Lightweight Cryptography, API, Embedded Systems, Security.

Abstract: The evolution of embedded systems and their applications in every daily activity, derive the development of lightweight cryptography. Widely used crypto-libraries are too large to fit on constrained devices, like sensor nodes. Also, such libraries provide redundant functionality as each lightweight and ultra-lightweight application utilizes a limited and specific set of crypto-primitives and protocols. In this paper we present the ULCL crypto-library for embedded systems. It is a compact software cryptographic library, optimized for space and performance. The library is a collection of open source ciphers (27 overall primitives). We implement a common lightweight API for utilizing all primitives and a user-friendly API for users that aren't familiar with cryptographic applications. One of the main novelties is the configurable compilation process. A user can compile the exact set of crypto-primitives that are required to implement a lightweight application. The library is implemented in C and measurements were made on PC, BeagleBone and MemSic IRIS devices. ULCL occupies 4 – 516.7KB of code. We compare our library with other similar proposals and their suitability in different types of embedded devices.

1 INTRODUCTION

OpenSSL and other well-known crypto libraries (The OpenSSL Project, 2013), target mainstream applications. Such libraries support high levels of security and don't take into consideration the special needs of constrained and ultra-constrained devices. Other libraries that are designed for embedded system applications support crypto-primitives that are optimized for space, speed or power consumption. On the other hand, such libraries either contain redundant functionality as they support a wide range of cryptographic primitives, like CyaSSL (WolfSSL Inc., 2013), or contain a small number of primitives, like Edon (Gligoroski, 2003).

OpenSSL (The OpenSSL Project, 2013) is designed for mainstream applications with high level of security and high speed. It provides a complete set of cryptographic functionality. On the other hand, OpenSSL hasn't embodied newer cryptographic standards, like TLS 1.2 and DTLS, and progressive primitives, like the eSTREAM (ECRYPT, 2008) finalists and the SHA-3 (NIST, 2012). The code organization is burden with legacy code and makes difficult its use by developers. For mainstream devices, OpenSSL achieves high en/decryption rates. It utilizes assembly code to speed up computationally heavy operations for several processors as well as the Intel AES-NI. For embedded devices, the executable code's size can be unacceptable. The library is open source.

CyaSSL (WolfSSL Inc., 2013) targets embedded and real-time operating system (RTOS) environments. It is a lightweight SSL library and supports new cryptographic standards and progressive ciphers. CyaSSL was written with developers in mind and provides a simple and documented API with easy-to-use abstraction layers for OS and custom I/O. It also supports an OpenSSL compatible API for broader use and acceptance. CyaSSL is the leading SSL library for embedded, real-time, mobile and enterprise systems. It achieves high performance while keeping small code size and low memory usage per connection. The library is licensed under the GPLv2 and a commercial license.

Hatzivasilis G., Theodoridis A., Gasparis E. and Manifavas C..
ULCL - An Ultra-lightweight Cryptographic Library for Embedded Systems.
DOI: 10.5220/0004900602470254
In *Proceedings of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems* (MeSeCCS-2014), pages 247-254
ISBN: 978-989-758-000-0
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

ULCL is appropriate for lightweight and ultra-lightweight applications where specific cryptographic primitives are required. It provides basic cryptographic functionality and supports progressive ciphers. The APIs achieve low overhead and comparable overall performance while remain easy-to-use even by developers that aren't familiar with cryptography. The size of the executable code is the smallest possible as the compilation is adjusted to the application scenario. The library is open source.

We apply OpenSSL, CyaSSL and ULCL on BeagleBone (BeagleBoard.org Foundation, 2011) devices with the default compilation options. All libraries are implemented in C and fair measurements were made. BeagleBone is a low-cost credit-card-sized embedded device that runs Ubuntu and connects with the Internet. It embodies an AM3359 ARM Cortex-A8 single core CPU running at 500-720 MHz.

2 RELATED WORK

Edon (Gligoroski, 2003) is an ultra-lightweight library for embedded systems. It is implemented in C and occupies about 5KB of memory. Edon uses quasigroups to build cryptographic primitives and develop a block cipher, a stream cipher, a hash function and a pseudorandom number generator.

The CACE Networking and Cryptography library (NaCl) (Bernstein, 2009) is an easy-to-use high-speed high-security public-domain library for network communication and cryptographic applications. The library provides a high level API – called crypto-box – for implementing public-key authenticated encryption. The user realizes the whole process as a single step and doesn't consider the internal parameters and communication steps between the participants. NaCl performs speed tests at compilation time and selects the best crypto-primitives for each device. A user can also use low-level APIs to apply specific primitives. Versions of the library are supported in C, C++ and Python. In C, the code occupies 17.36 – 27.96KB of memory (Hutter and Schwabe, 2013)

In section 3, we describe the Ultra-Lightweight Crypto-Library (ULCL) for embedded systems, the main concepts and the measurements on real devices. In section 4, we compare our proposal with other libraries. In section 5, we conclude.

3 ULCL

We implement the ULCL for the cryptographic technologies of the node layer. The library provides 'built in' cryptographic functionalities for embedded systems that make use of a specific set of cryptographic primitives and protocols. It utilizes open source ciphers' implementations, two lightweight APIs and a configurable compilation process.

Only block/stream ciphers and hash functions are included. The library provides basic cryptographic functionality for constrained and ultra-constrained devices. It targets on application environments where asymmetric cryptography can't be applied. As asymmetric cryptography is much more resource demanding than symmetric one, these applications depend on dependable authentic key distribution mechanisms (Chen and Chao, 2011). Such mechanisms are lightweight key management solutions that utilize only symmetric cryptography.

We consider two types of embedded devices. The BeagleBone (BeagleBoard.org Foundation, 2011) is a constrained device with 500 MHz processing power, 256 MB memory and Ubuntu Linux operating system. We perform the basic measurements of ULCL, Edonm NaCl, CyaSSL and OpenSSL on such devices. The Memsic IRIS (Memsic Inc., 2010) is an ultra-constrained embedded device with 8MHz processing power, 8KB memory and Contiki operating system. CyaSSL and OpenSSL don't fit on such devices. We apply our library on IRIS as a proof of concept that ULCL is appropriate for ultra-constrained devices and applications. The measurements that are reported in subsection 3.1 were performed on a PC with Intel core 2 duo e8400 (3GHz), 2GB of RAM and Linux operating system.

3.1 Open Source Cipher Implementations

ULCL utilizes open source implementations of known ciphers. It is a collection of lightweight or compact implementations of standard block/stream ciphers and message authentication code (MAC) primitives. In (Manifavas et al., 2013), all these primitives are evaluated and the best of them are proposed for different types of embedded devices.

For block ciphers, it supports AES (Erdelsky, 2002), DES/3DES (CIFS Library, 2010), PRESENT (Klose, 2007), LED (Guo et al., 2011), KATAN/KTANTAN (Canniere et al., 2009), Clefia (SONY, 2008), Camellia (NTT, 2013), XTEA

(Wheeler and Needham, 1997) and XXTEA (Wheeler and Needham, 1998). AES and Camellia are designed for mainstream applications with high level of security and throughput. Their space requirements are high for ultra-constrained devices and are included only for more powerful embedded devices. DES, 3DES and Clefia are designed for lightweight cryptography on embedded systems and support high and moderate level of security, lower throughput and consume less computational resources. They are appropriate for constrained devices. PRESENT, XTEA and XXTEA are designed for ultra-constrained devices with even lower level of security and throughput and are efficient in power-energy-memory. LED, KATAN and KTANTAN are mainly implemented in ultra-constrained hardware and their performance in software is low. We include them for higher compatibility in heterogeneous systems where nodes may use hardware-accelerated cryptography.

All block ciphers operate in ECB, CBC and CTR modes of operation (Dworkin, 2001). Furthermore, the library supports all known padding schemes: zeroPadding, PKCS5, PKCS7, ISO_10126-2, ISO_7816-4 and X9.23.

Table 1, summarizes the block ciphers' features. The ciphers are executed in ECB node of operation with zero padding. Figure 1, illustrates the relevant features of the ciphers with 128-bit key.

Table 1: Block ciphers in ECB mode of operation with zero padding.

Cipher	Key (bits)	Block (bits)	Code (KB)	RAM (KB)	Throughput (MBps)
AES	128	128	25	10.25	56.35
AES	192	128	25	10.33	50.25
AES	256	128	25	10.41	69.82
3DES	64	64	12	10.02	20.15
3DES	128	64	12	10.07	20.25
3DES	192	64	12	10.11	20.21
Camellia	128	128	33	10.08	68.31
Camellia	192	128	33	10.13	55.57
Camellia	256	128	33	10.18	55.17
Clefia	128	128	6.9	10.08	4.65
Clefia	192	128	6.9	10.13	3.83
Clefia	256	128	6.9	10.17	3.29
XTEA	128	64	2.7	10.06	26.07
PRESENT	80	64	2.6	14.46	0.44
PRESENT	128	64	2.6	14.50	0.44

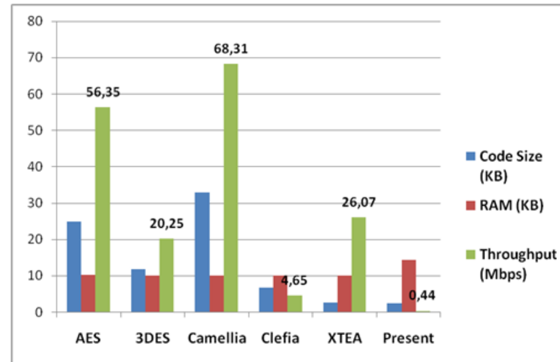

Figure 1: ULCL block ciphers with 128-bit key.

For stream ciphers, ULCL supports ARC4 (CyASSL, 2013) and the eSTREAM project (ECRYPT, 2008) finalists Salsa20 (Bernstein, 2005), Rabbit (Boesgaard et al., 2005), HC128 (Wu, 2005), SOSEMANUK (Berbain et al., 2005), Grain (Hell, M., Johansson, T. & Meier, W., 2005), Trivium (Canniere and Preneel, 2005) and MICKEY v2 (Babbage and Dodd, 2005). RC4 is the most widely used stream cipher. It achieves high throughput but it is considered insecure for new applications. The finalists Salsa20, Rabbit, HC128 and SOSEMANUK are designed for software. They achieve higher throughput and are considered secure against all attacks faster than the exhaustive search. Salsa20 and Rabbit are the most attractive for constrained devices. HC128 utilizes two large tables to perform en/decryption. Due to its table-driven approach it is very fast but requires much memory. The finalists Grain, Trivium and MICKEY v2 are designed for hardware but perform reasonable in

Table 2: Stream ciphers.

Cipher	Key / IV (bits)	Code (KB)	RAM (KB)	Throughput (MBps)
HC128	128/ 128	7.9	16.58	517.60
Rabbit	128 / 64	3.1	8.41	264.29
Salsa20	128 / 64	3.1	8.27	155.10
Salsa20	256 / 64	3.1	8.35	154.97
SOSEMA NUK	128 / 128	15	9.07	300.50
SOSEMA NUK	256 / 128	15	9.14	299.03
Grain	80 / 64	2.7	16.26	64.24
Grain-128	128 / 96	3	24.4	91.39
MICKEY v2	80 / 80	3.2	8.22	2.85
Trivium	80 / 80	6.9	8.25	180.43
ARC4	40 / 0	1.22	8.55	165.69

ARC4	2048 / 0	1.22	9.78	164.50

software. They were also cryptanalyzed and found secure. Table 2 and Figure 2, summarizes the stream ciphers' features.

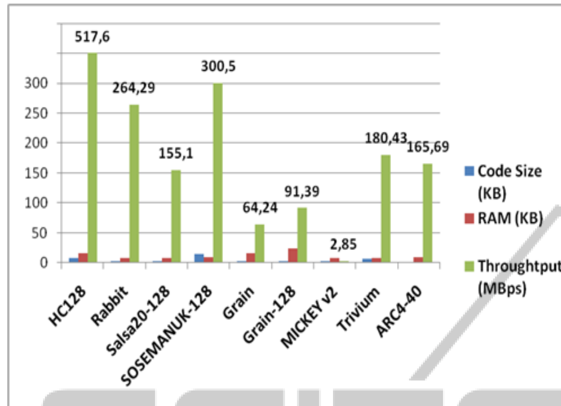

Figure 2: ULCL stream ciphers.

Table 3: MACs.

Hash	Digest (bits)	Code (KB)	RAM (KB)	Throughput (MBps)
Blake	224	17	2.47	94.86
Blake	256	17	2.48	95.66
Blake	384	17	2.53	36.63
Blake	512	17	2.57	36.91
Groestl	224	29	2.46	29.62
Groestl	256	29	2.47	29.82
Groestl	384	29	2.83	20.08
Groestl	512	29	2.87	20.13
JH	224	7.7	2.32	17.26
JH	256	7.7	2.33	17.26
JH	384	7.7	2.38	17.27
JH	512	7.7	2.42	17.25
Keccak	224	68.1	2.54	62.12
Keccak	256	68.1	2.55	62.71
Keccak	384	68.1	2.6	52.57
Keccak	512	68.1	2.65	36.80
Skein	256	52.4	2.39	61.51
Skein	512	52.4	2.48	61.35
Skein	1024	52.4	2.67	46.37
MD5	128	3.3	2.15	308.64
SHA-1	160	6	2.17	167.85
SHA256	256	3.7	2.22	95.44
SHA512	512	17	2.41	42.82

For MACs, ULCL supports MD5 (CyASSL, 2013), SHA-1 (CyASSL, 2013), SHA-2 (CyASSL, 2013), SHA-3 (Keccak) (Bertoni et al., 2008) and the other SHA-3 contest's (NIST, 2012) finalists Blake (Aumasson et al., 2008), JH (Wu, 2008), Groestl (Gauravaram et al., 2008) and Skein (Ferguson et al., 2008). MD5 and SHA-2 are the most known MACs for mainstream applications. MD5 isn't collision resistant, thus less secure, but is very fast. SHA-2 is still secure and the SHA-3 contest targeted to establish an alternative standard. The SHA-3 functions are newer progressive MACs that adopt different design features than SHA-2. Table 3 and Figure 3, summarizes the MACs' features.

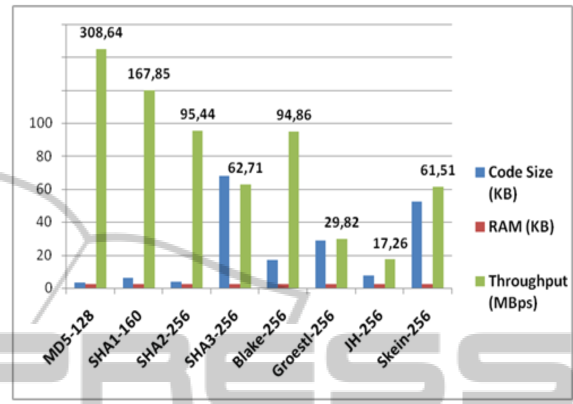

Figure 3: ULCL MACs.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

3.2 Lightweight APIs

For every crypto-primitive type (block cipher, stream cipher, MAC), we implement a common API for utilizing all the different primitives with their parameters. The API was designed with developers in mind and is easy-to-use. There are common functions in each category for initialization and processing. All size values are measured in bytes.

The common API for the *block ciphers* is:

`ulcl_block_ctx *ulcl_block_init (char *cipher, byte *key, int key_size):` Initializes the block cipher (cipher) and the encryption key (key). The function returns a cipher data structure.

`int ulcl_block_destroy (ulcl_block_ctx *ctx):` Frees the allocated memory. The function returns an error code.

`byte *ulcl_block_encrypt (ulcl_block_ctx *ctx, byte *plaintext, int pl_size, char *mode, char *padding):` Encrypts a plaintext (plaintext) with the block cipher (ctx) and the mode of operation (mode) and padding (padding). The function returns the ciphertext.

`byte *ulcl_block_decrypt (ulcl_block_ctx *ctx, byte *ciphertext, int cp_size):` Similar with 'ulcl_block_encrypt'. The function decrypts a ciphertext and returns the plaintext.

The common API for the *stream ciphers* is:

`ulcl_stream_ctx *ulcl_stream_init (char *cipher, byte *key, int key_size, byte *iv, int iv_size)`: Initializes the stream cipher (cipher) and the encryption key (key) and the IV (iv). The function returns a cipher data structure.

`int ulcl_stream_destroy (ulcl_ stream_ctx *ctx)`: Frees the allocated memory. The function returns an error code.

`byte *ulcl_stream_encrypt (ulcl _stream_ctx *ctx, byte *plaintext, int pl_size)`: Encrypts a plaintext (plaintext) with the stream cipher (ctx). The function returns the ciphertext.

`byte *ulcl_stream_decrypt (ulcl _stream_ctx *ctx, byte *ciphertext, int cp_size)`: Similar with 'ulcl_stream_encrypt'. The function decrypts a ciphertext and returns the plaintext.

The common API for the *MACs* is:
`ulcl_hash_state *ulcl_hash_init (char *hash_func, int hash_size)`: Initializes the hash function (hash_func) and returns a hash function structure.

`int ulcl_hash_destroy (ulcl_ hash_state *state)`: Frees the allocated memory. The function returns an error code.

`byte *ulcl_get_hash (ulcl_hash_state *state, byte *msg, int msg_size)`: Processes the message (msg) with the hash function (state). The function returns the digest.

Moreover, we implement a second API for developers that aren't familiar with cryptography. The developers don't deal with selecting crypto-primitives and their parameters. When this simple API is compiled, ULCL performs a test program to figure out the system's capabilities in memory and processing capabilities. According to this test, ULCL invokes internally the most appropriate primitives for this device. The API is suitable for cryptographic applications in homogeneous systems. Also, it is appropriate for educational purposes in computer security. All the function arguments are strings and contain hexadecimal numbers in string form.

The simple API for the *block ciphers* is:
`char *ulcl_block_encrypt_simple (char *data, char *key)`: Encrypts a plaintext (data) with the key (key) and returns the ciphertext.

`char *ulcl_block_decrypt_simple (char *data, char *key) )`: Decrypts a ciphertext (data) with the key (key) and returns the plaintext.

The simple API for the *stream ciphers* is:

`char *ulcl_stream_encrypt_simple (char *data, char *key, char *iv)`: Encrypts a plaintext (data) with the key (key) and the IV (iv) and returns the ciphertext.

`char *ulcl_stream_decrypt_simple (char *data, char *key, char *iv)`: Decrypts a ciphertext (data) with the key (key) and the IV (iv) and returns the plaintext.

The simple API for the *MACs* is:
`char *ulcl_hash_simple(char *data)`: Processes the message (data) and returns the digest.

From the supported primitives, PRESENT, Clefia, XTEA, Rabbit, Salsa20, Grain, ARC4, MD5 and SHA-2 are the most compact (Figure 4).

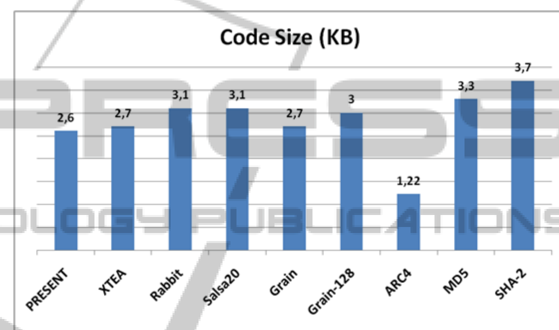

Figure 4: The most compact cryptographic primitives.

3DES, MICKEY v2, Trivium, JH and Skein require the least RAM (Figure 5).

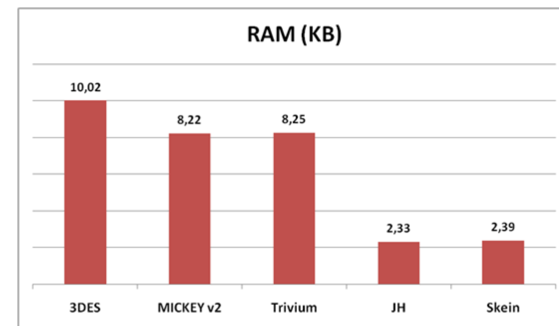

Figure 5: The cryptographic primitives that require the least RAM.

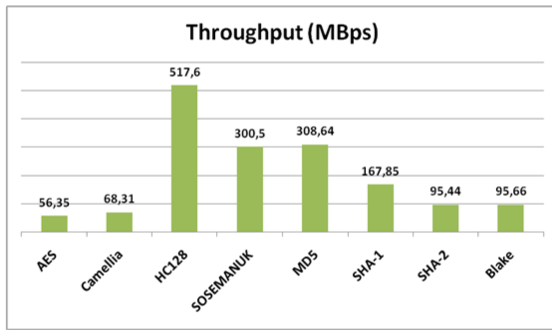

Figure 6: The fastest cryptographic primitives.

AES, Camellia, HC128, SOSEMANUK, MD5, SHA-1, SHA-2 and Blake are the fastest (Figure 6).

3.3 Configurable Compilation

Our main novelty is the configurable compilation process. A user can define an exact set of crypto-primitives that are compiled without compiling the whole library. Thus, the executable code that runs on the embedded device is small. For example a user can compile only a compact AES implementation and use it through the common API for block ciphers. As embedded devices run a specific set of protocols and crypto-primitives, our configurable implementation of lightweight primitives is a good candidate library.

The block ciphers API occupies 5.22 – 23.5KB of ROM memory. The overhead is high as we implement the modes of operation and the padding schemes except from the API for en/decryption. The stream ciphers API occupies 1.74 – 7.4KB and the hash functions API occupies 1.75 – 7.4KB. The total API overhead is about 1.74 – 38.3KB and the library occupies 4 – 516.7KB.

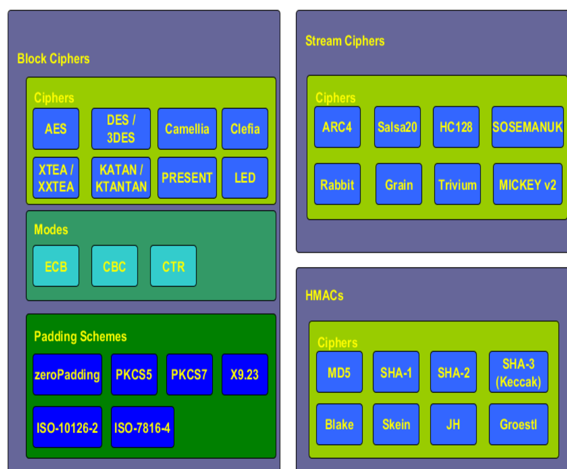

Figure 7: Segmented compilation.

Figure 7, illustrates the segmented compilation process. Each box represents a different compilation option. For example, a user can compile the whole library, the block ciphers or specific crypto-primitives.

3.4 Other Features

ULCL is open source and well-documented. The library contains a series of examples and test files. The examples demonstrate the compilation process capabilities and the utilization of the APIs. A user can execute command line tests for each compiled primitive. Furthermore, a benchmark suite is provided for measuring the library's features in the application setting. The APIs performs extensive error checking and reports relative error codes. The correct functionality of the whole library is validated. Each crypto-primitive is verified through the manufacturer's test vectors and common tests with well-known libraries, like OpenSSL.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

4 DISCUSSION

ULCL's ordinary compilation occupies the least code size as it offers only basic cryptographic functionality. By default OpenSSL installs the full library while CyaSSL installs only a core set of it. CyaSSL can occupy about 90% less code size than OpenSSL. The code footprint is a significant factor for the applications that we study. ULCL's size is suitable for lightweight and ultra-lightweight applications and CyaSSL's size is suitable for lightweight ones. OpenSSL produce high footprint, which is over 1MB. Figure 8, illustrates the code size of the examined libraries.

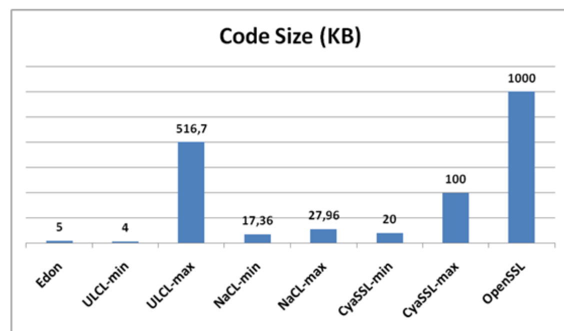

Figure 8: The code size of the examined libraries.

In comparison with Edon, ULCL supports a variety of known standard or lightweight ciphers, whose security properties are well-studied, while keeping

an easy-to-use API. For the user-friendly API and the internal selection of the best primitives, ULCL uses a similar testing procedure at compilation time as NaCl.

5 CONCLUSIONS

We implement the compact crypto-library ULCL for constrained and ultra-constrained devices. The library embodies open source implementations of known ciphers and provides a variety of basic crypto-primitives. Totally, 27 cryptographic primitives are supported as well as 6 padding schemes and 3 modes of operation for the block ciphers. The user can configure the compilation process and compile the exact set of primitives that are necessary, without redundant functionality. ULCL requires 4 – 516.7KB of code. For each cryptographic primitive type (block/stream ciphers and hash functions), there is a common API for all the relevant primitives and an API for users that aren't familiar with cryptography. ULCL is well-documented, with several example and test programs, and its correct operation is validated.

ACKNOWLEDGEMENTS

This work was funded by the General Secretarial Research and Technology (G.S.R.T.), Hellas under the Artemis JU research program nSHIELD (new embedded Systems arcHItecturE for multi-Layer Dependable solutions) project. Call: ARTEMIS-2010-1, Grand Agreement No: 269317.

REFERENCES

Aumasson, J.-P. et al., 2008. Blake implementation. SHA-3 contest. Available at: http://131002.net/blake/ .

Babbage, S. & Dodd, M., 2005. MICHEY v2 implementation. eSTREAM project. Available at: http://www.ecrypt.eu.org/stream/e2-mickey.html .

BeagleBoard.org Foundation, 2011. BeagleBone manual. Available at: http://beagleboard.org/bone .

Berbain, C. et al., 2005. SOSEMANUK implementation. eSTREAM project. Available at: http:// www.ecrypt.eu.org/stream/e2-sosemanuk.html .

Bernstein, D. J., 2009. Cryptography in NaCl. Networking and Cryptography library, (Mc 152). Available at: http://cr.yp.to/highspeed/naclcrypto-20090310.pdf .

Bernstein, D. J., 2005. Salsa20 implementation. eSTREAM project. Available at: http:// www.ecrypt.eu.org/stream/e2-salsa20.html .

Bernstein, D. J., Lange, T. & Schwabe, P., 2012. The security impact of a new cryptographic library. Progress in Cryptology– LATINCRYPT 2012, Springer Berlin Heidelberg, LNCS, 7533, pp.159–176. Available at: http://link.springer.com/ chapter/10.1007/978-3-642-33481-8_9 .

Bertoni, G. et al., 2008. SHA-3 (Keccak) implementation. SHA-3 contest. Available at: http:// keccak.noekeon.org/ .

Boesgaard, M. et al., 2005. Rabbit implementation. eSTREAM project. Available at: http:// www.ecrypt.eu.org/stream/e2-rabbit.html .

Canniere, C. de, Dunkelman, O. & Knezevic, M., 2009.

KATAN/KTANTAN implementation. Available at: http://www.cs.technion.ac.il/~orrd/KATAN/ .
Canniere, C. De & Preneel, B., 2005. TRIVIUM implementation. eSTREAM project. Available at: http://www.ecrypt.eu.org/stream/e2-trivium.html .
Chen, C.-Y. & Chao, H.-C., 2011. A survey of key distribution in wireless sensor networks. In J. & S. Wiley, ed. Security and Communication Networks. Wiley Online Library, p. n/a–n/a. Available at: http://doi.wiley.com/10.1002/sec.354 .
CIFS Library, 2010. DES & 3DES implementation. Available at: http://www.ubiqx.org/proj/ libcifs/source/Auth/ .
CyASSL, 2013. MD5, SHA1, SHA256, SHA512, ARC4 implementatons. Available at: http://www.yassl.com/ yaSSL/Products-cyassl.html .
Dworkin, M., 2001. Recommendation for block cipher modes of operation, NIST special publication, Available at: http://csrc.nist.gov/publications/ nistpubs/800-38a/sp800-38a.pdf .
ECRYPT, 2008, eSTREAM project. Available at: http://www.ecrypt.eu.org/stream/.
Erdelsky, P.J., 2002. AES implementation. Available at: http://www.efgh.com/software/rijndael.htm .
Ferguson, N. et al., 2008. Skein implementation. SHA-3 contest. Available at: http://www.skein-hash.info/ .
Gauravaram, P. et al., 2008. Groestl implementation. SHA-3 contest. Available at: http://www.groestl.info/ .
Gligoroski, D., 2003. Edon-library of reconfigurable cryptographic primitives suitable for embedded systems. In Workshop on cryptographic hardware and embedded systems. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10. 1.1.13.7586&rep=rep1&type=pdf .
Guo, J. et al., 2011. LED implementation. Available at: http://sites.google.com/site/ledblockcipher/home-1 .
Hell, M., Johansson, T. & Meier, W., 2005. Grain implementation. eSTREAM project. Available at: http://www.ecrypt.eu.org/stream/e2-grain.html .
Hutter, M. & Schwabe, P., 2013. NaCl on 8-bit AVR Microcontrollers. IACR Cryptology ePrint Archive2. Available at: http://eprint.iacr.org/2013/375.pdf.
Klose, D., 2007. PRESENT implementation. Available at: http://www.lightweightcrypto.org/implementations.ph p .
Manifavas, C. et al., 2013. Lightweight Cryptography for Embedded Systems - A Comparative Analysis. 6th International Workshop on Autonomous and Spontaneous Security – SETOP2013, Springer, LNCS, 8247, pp.1–18.
Memsic Inc., 2010. Memsic Isis manual. Available at: http://www.memsic.com/userfiles/files/User-Manuals/iris-oem-edition-hardware-ref-manual-7430-0549-02.pdf .
NIST, 2012. SHA-3 contest. Available at: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html .
NTT - Secure Platform Laboratories - Information Security Project, 2013. Camellia implementation. Available at: https://info.isl.ntt.co.jp/crypt/ eng/camellia/ .
SONY, 2008. Clefia implementation. Available at: http://www.sony.net/Products/cryptography/clefia/ .
The OpenSSL Project, 2013. Openssl, Available at: http://www.openssl.org .
Wheeler, D. & Needham, R., 1997. XTEA implementation. Available at: http://en.wikipedia.org/ wiki/XTEA .
Wheeler, D. & Needham, R., 1998. XXTEA implementation. Available at: http://en.wikipedia.org/ wiki/XXTEA .
WolfSSL Inc., 2013. CyaSSL embedded SSL library. Available at: http://yassl.com/yaSSL/Products-cyassl.html .
Wu, H., 2005. HC-128 implementation. eSTREAM project. Available at: http://www.ecrypt.eu.org/ stream/e2-hc128.html .
Wu, H., 2008. JH implementation. SHA-3 contest. Available at: http://www3.ntu.edu.sg/home/ wuhj/research/jh/ .

SCIENCE AND TECHNOLOGY PUBLICATIONS