Implementation of a New Cipher in OpenSSL Environment the Case of INDECT Block Cipher

Piotr Jurkiewicz, Marcin Niemiec*
AGH University of Science and Technology, Department of Telecommunications, al. A. Mickiewicza 30, 30-059 Krakow, Poland.

* Corresponding author. Tel.: +48 126174803; email: niemiec@kt.agh.edu.pl
Manuscript submitted November 24, 2014; accepted April 8, 2015.
doi: 10.17706/ijcce.2016.5.1.41-49

Abstract: Implementation of a new cipher in the popular cryptographic library is a convenient way of deploying it for wide usage. In this article we described the process of integration of a new cipher with the OpenSSL libraries. In this case it is the INDECT Block Cipher (IBC), a new symmetric block cipher based on a substitution-permutation network. However, this work is more universal because it is possible in the same way to integrate any symmetric cipher in the OpenSSL environment. The goal of the implementation is to enable usage of a new cipher in applications using OpenSSL libraries, especially to encrypt data in SSL/TLS connections. The article contains general descriptions of OpenSSL, SSL/TLS protocol and the IBC algorithm. Also, the integration process was described in detail. Particular attention is paid to binary compatibility issues. Additionally, results of various tests of the modified libraries were presented.

Key words: Cryptography, INDECT block cipher (IBC), OpenSSL, transport layer security (TLS).

1. Introduction

Developing a new cipher is exhausting work. The created algorithm should be not only be secure, but must also achieve a high performance. Therefore, many tests and theoretical analyses must be performed before the new algorithm's specifications can be released. At the end of the process comes the point where the new cipher is ready. However, the struggle doesn't end there. The new algorithm should be properly implemented, in order to use it in the real world.

The implementation would be most useful if it would enable usage of a new cipher in already existing applications. The most convenient way to do this is to implement a new algorithm in some widely used cryptographic library.

There are a few popular cryptographic libraries in use nowadays: crypto++, Mozilla NSS, OpenSSL, and others. In this case study, the OpenSSL toolkit was chosen as the base for new cipher implementation because of its popularity.

This article consists of 7 sections. Section 2 describes the structure of the OpenSSL toolkit and contains an overview of the SSL/TLS protocol. Section 3 describes the implemented algorithm — the INDECT Block Cipher. Section 4 presents details about integration of a cipher. Section 5 discusses compatibility issues. Results of tests performed on modified libraries are presented in Section 6. Section 7 contains the summary.

2. OpenSSL Toolkit

OpenSSL [1] is an open source toolkit implementing SSL/TLS protocols as well as general purpose cryptography functions. It is written in the C programming language. As well as the toolkit being used as a stand-alone cryptographic application, it provides libraries which can be reused by independent programs.

OpenSSL distributions are available for most operating systems, including Unix-like ones, Windows and MacOS. OpenSSL shared libraries are preinstalled in many Linux distributions. It consists of three major components: two libraries (*libcrypto* and *libssl*) and the command line tool (*openssl*).

2.1. Cryptographic Features

The *libcrypto* library implements a wide range of cryptographic-related functions, including: symmetric encryption, public key cryptography, certificate handling and hash functions.

The library has several built-in symmetric encryption algorithms. It includes AES, DES, Blowfish, IDEA and others. Most of the ciphers can work in block and stream modes. OpenSSL supports the following modes:

Block modes:

- ECB (Electronic Codebook)
- CBC (Cipher Block Chaining)

Stream modes:

- CFB (Cipher Feedback)
- OFB (Output Feedback)
- CTR (Counter Mode)

The *libcrypto* library provides an API which allows these ciphers to be used to encrypt or decrypt data in applications using the library. Moreover, the *openssl* command-line tool can encrypt or decrypt arbitral files, using provided keys or keys based on passwords.

2.2. Secure Communication

SSL (Secure Sockets Layer) [2] and its successor, TLS (Transport Layer Security) [3] are network protocols that provide secure communication over the Internet. They integrate the data cryptography functions, allowing client/server applications to protect against eavesdropping and tampering.

SSL/TLS is usually implemented on top of some transport protocol, e.g. Transmission Control Protocol (TCP). It encapsulates the application data. A big advantage of SSL/TLS is that it is application protocol independent. A higher level protocol can layer on top of the SSL/TLS protocol transparently.

SSL and TLS use symmetric ciphers (e.g. AES) for data encryption. The keys for these ciphers are generated uniquely for each connection and are based on a secret negotiated by the Handshake Protocol. SSL/TLS supports many combinations of ciphers, authentication mechanisms, and hashing algorithms. These combinations are called ‘ciphersuites’. Ciphersuites are chosen during the negotiation.

The *libssl* library implements the SSL and TLS protocol handling. The library can be used by third-party applications to secure their connections using SSL/TLS protocols. The *libssl* provides a rich API. A developer can create either a server or a client application using the library’s functions.

The *libssl* library makes use of cryptographic functions provided by the *libcrypto* library. In particular, it uses symmetric ciphers to encrypt data transmitted in SSL/TLS connections.

3. INDECT Block Cipher

The INDECT Block Cipher (IBC) is a new symmetric block cipher. The paper [4] describes this algorithm in detail. The construction of the cipher is based on a substitution-permutation network. The main innovation of this symmetric algorithm is the unique approach to the key scheme. The algorithm uses highly non-linear S-boxes and P-boxes, generated from the key. IBC provides high resistance to cryptanalysis and wide key space, up to 576 bits.

3.1. Structure of IBC

IBC works on 256-bit blocks of data. The block of data is initially permuted. Then, a single round loop begins. The number of rounds depends on the key size. The overall structure of a single round is presented in Fig. 1.

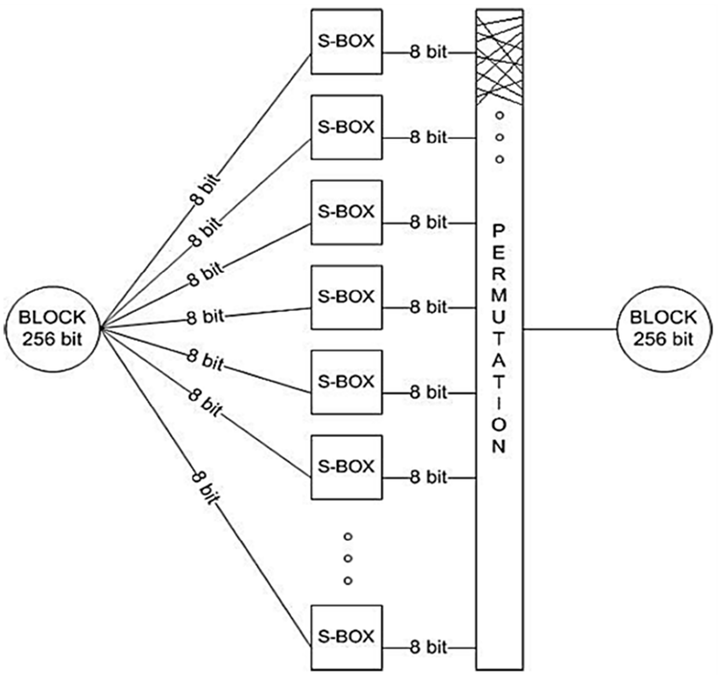


Fig. 1. IBC round structure [4].

Each round consists of a substitution and a permutation phase. In the substitution phase, the algorithm splits the 256-bit block into 8-bit sub-blocks. Then each byte is substituted by a S-box.

The number of different S-boxes used for substitution depends on the key size. For 128-bit key, all 32 sub-blocks are substituted by the same S-box. However, two different S-boxes are used for a 192-bit key: the first half of sub-blocks are substituted by the first S-box, and the second half of sub-blocks are substituted by the second S-box. Similarly, for a 320-bit key four S-boxes are used, and for 576-bit keys eight S-boxes are used.

After the substitution, all sub-blocks are merged back into one 256-bit block. Then, the whole block is permuted. Permutation is based on another S-box, called the P-box. The number of the bits in the 256-bit block is substituted in the P-box, and the outcome number marks the destination spot. This means that bits are shuffled within the whole 256-bit block.

3.2. S-box Generation

Unlike in many other ciphers, data is not XORed with key-dependent round keys in each round. Instead, the substitution and permutation processes depend on the key itself. New S-boxes are created from the original AES S-box by a linear transformation.

Using this method, we are able to create a huge number of new S-boxes. The maximum number of S-boxes that we can obtain this way is:

$$5348063769211699200 \approx 5.35 \times 10^{18}$$

Each one of the S-boxes can be coded on 64 bits. The original method of constructing new S-boxes was tested in a specially developed S-box simulator. It has been confirmed that generated S-boxes have good security features: balancing, SAC, completeness, diffusion order, low XOR table, and nonlinearity. Further details about new cipher construction and its verification can be found in [4].

4. Integration

Before the implementation of a new cipher, several requirements were set up:

- The new cipher should be available in all functions of OpenSSL where other symmetric ciphers are used;
- The new cipher should be able to encrypt data transmitted in SSL/TLS connections;
- The performance of encryption should be as high as possible;
- The deployment of modified libraries should be as simple as possible;
- Usage of a new cipher in existing applications should not require modification of these applications or recompilation.

However, although OpenSSL is very popular in network environments, there is unfortunately no tutorial on integration of new ciphers. Moreover, OpenSSL's developing documentation is rather poor. Therefore, the authors decided to take a look at an existing cipher implementation and try to implement a new one similarly.

The Camellia cipher [5] was chosen as an example. First, OpenSSL source code files were searched for appearance of the words "camellia" and "cmll". There were 2138 hits in 62 files. After code analyzing we established that implementation should consist of two main parts: implementation of a cipher in *libcrypto* library and implementation of SSL/TLS ciphersuite in *libsssl*.

4.1. Implementation of a Cipher

Ciphers' code is a part of *libcrypto* library. The code is located in crypto/x directories, where *x* is the name of an individual algorithm.

Every cipher directory contains a main header file, with the same name as a cipher directory. The main header file contains declarations of structures and functions. Names of functions consist of a prefix, usually the same as the name of the cipher. Functions' prototypes are identical for every cipher, providing the interface to the underlying cipher.

New functions: *Indect_set_encrypt_key* and *Indect_set_decrypt_key* are responsible for setting up encryption/decryption options and generating internal keys based on settings and keys provided by the user. Functions take strings containing the encryption key provided by the user as an input. The output of functions is stored in the *INDECT_KEY* type structure.

This structure is cipher-dependent, so it can be different for every algorithm. It stores all the key information needed by the low level encryption/decryption functions. In the case of the IBC, the structure contains S-Boxes generated from the key during the key initialization phase as well as permutation tables, derived from the P-Box.

Another function: *Indect_encrypt* and *Indect_decrypt* are responsible for encrypting and decrypting a single block of data. Because the IBC uses different encryption/decryption routines for each key length, these functions cannot directly implement block encryption/decryption. Instead, these functions invoke one of the actual encryption functions indirectly, by dereferencing function pointer arguments stored in the key structure.

Next, the main header file contains declarations of functions implementing modes of operation (ECB, CBC, OFB, etc.). These functions do not encrypt/decrypt data itself. They only handle the data and prepare it for processing (partitioning into blocks, padding blocks). To encrypt/decrypt a single block they call *Indect_encrypt* and *Indect_decrypt* functions, described above. Thus, functions implementing block modes are mostly cipher invariant. Implementation of these functions can be borrowed from ciphers existing in OpenSSL.

Functions performing actual key setup and encryption/decryption are declared locally in separate files. Therefore, these functions' interface does not depend on OpenSSL API. Functions are declared *ibc_locl.c* and are implemented in the *indect.c* file.

4.2. Integration with Libcrypto

Integrating a new cryptographic algorithm requires a number of modifications to the OpenSSL code. Several *libcrypto* functions, which implement individual functionalities, must know all available ciphers. New algorithm modes should be properly registered and calls to cipher's routines should be added into the *libcrypto* code.

The *crypto/evp/* directory stores the code implementing the EVP library. The EVP library provides a high-level interface to cryptographic functions. It requires several modifications in order to add a new algorithm.

- First, the *crypto/evp/c_allc.c* file should be modified. It contains calls to functions registering all cipher algorithms. Calls registering a new algorithm need to be added.
- Second, the *crypto/evp/evp.h* header file should be modified. The value of the constant variable defining the maximum key length should be modified. Originally it is 32 bytes (256 bits) — in case of IBC its needs to be changed to 40 bytes (320 bits) because IBC can use a 320-bit key.

The *crypto/evp/evp.h* ends with error codes. After the modification, it contains error codes related to IBC. They are automatically generated by the script mkerr.pl and should not be modified manually. Similarly, the *crypto/evp/evp_err.c* file contains error codes. This file is also automatically generated, and should not be modified manually. In order to generate error codes in both files it is needed to call the *make errors* command in the main source directory after applying all modifications in the source.

4.3. Implementation of SSL/TLS Ciphersuite

In order to make the new cipher available in SSL/TLS connections there must be several modifications applied in the *libssl* library. New ciphersuites containing the new cipher must be created and registered.

Ciphersuites are combinations of authentication, encryption, and Message Authentication Code (MAC) algorithms used to negotiate the security settings for a network connection using the SSL/TLS network protocol. The structure and use of the ciphersuite concept is defined in the document [3].

The ciphersuites' IDs are assigned by IANA and defined in several RFCs. There is a range of ciphersuite IDs reserved for private use (first byte of ciphersuite ID equals 'FF'). Unless a programmer is implementing a ciphersuite with an ID assigned for IANA, he/she should choose an ID from that range. Ciphersuite IDs are defined in the *ssl/tls1.h* header file.

The ciphersuites are implemented in the *ssl/s3_lib.c* file. Each ciphersuite is implemented in a structure which contains all settings related to that particular ciphersuite.

```
/* Cipher FF41 */
{
1,
TLS1_TXT_RSA_WITH_INDECT_128_CBC_SHA,
TLS1_CK_RSA_WITH_INDECT_128_CBC_SHA,
SSL_kRSA|SSL_aRSA|SSL_INDECT|SSL_SHA|SSL_TLSV1,
SSL_NOT_EXP|SSL_HIGH,
0,
128,
128,
SSL_ALL_CIPHERS,
SSL_ALL_STRENGTHS
},
```

The next file which needs to be modified is the *ssl/ssl.h* header file. This contains definitions of several constants. When a new cipher is implementing, the constant specifying its name must be defined.

Furthermore, the *ssl/ssl.h* file contains the SSL_DEFAULT_CIPHER_LIST constant. This defines the default ciphersuites order of preference. The cipher list consists of one or more cipher strings separated by colons.

To enable preferred usage of IBC ciphersuites, we do not have to modify this constant, because the list is sorted in order of ciphers' key length. The IBC ciphersuite uses 320-bit key length, so it will be preferred over the AES ciphersuites (maximum 256-bit key length). It is worth mentioning that modifying this constant breaks the binary compatibility.

5. Compatibility

Most applications using OpenSSL use it as a dynamic library. In the Windows environment *libcrypto* and *libssl* dynamic library (**.dll*) files are usually distributed bundled with the application. The *dll* files are usually located in the application's main directory.

In the Linux environment, however, most applications use system wide shared libraries, located in */lib* or a similar system directory. Files are usually named libcrypto.so.x.x.x and libssl.so.x.x.x.

One of the main reasons behind integrating the IBC into OpenSSL was to enable the easiest possible means of applying the new cipher in existing applications. The most convenient way to do this would be to just replace existing shared library files with the modified ones. However, this is possible only when the binary compatibility is ensured between the original and modified libraries.

Backward binary compatibility is a feature of the new version of a library compared with an old version of the same library to guarantee that applications working with the old version keep working correctly with the new version without recompilation. Examples of changes resulting in breaking backward binary compatibility include changing data type size of a function parameter or changing the structure of a virtual table in a class. Such changes result in incorrect run-time behavior or even crashes of applications that use the corresponding function or class [6].

Therefore, binary compatibility needs to be kept in mind while making changes in the source code. During the integration of IBC into the OpenSSL we made efforts to keep the binary interface compatible. Thanks to that, it is possible to deploy the new cipher on existing systems without any requirement to change the code of applications using OpenSSL, or even without the need for recompiling them against the modified libraries.

The binary compatibility has been tested with an automatic tool called *abi-compliance-checker* [7]. It is a free, open source tool, available for Linux-like operating systems. The tool can check all types of changes that cause backward binary compatibility problems.

6. Tests

The implemented IBC cipher was tested in a network environment. The performance of the improved source-code of the new cipher was checked. Also, secure SSL/TLS connections with IBC algorithm were verified.

6.1. Performance

Nowadays, performance is a primary requirement for symmetric encryption. Therefore, we made efforts to optimize performance during the implementation of the new cipher in OpenSSL. The first implementations of IBC — simple graphical application and hardware implementation — were not very fast [8]. This is why the code of IBC was completely rewritten. Thanks to this, a significant improvement was archived.

The first improvement was migrating to the lower level programming language (from C# to C). The second was rewriting the permutation code using only low-level bit instructions (bit shift, OR).

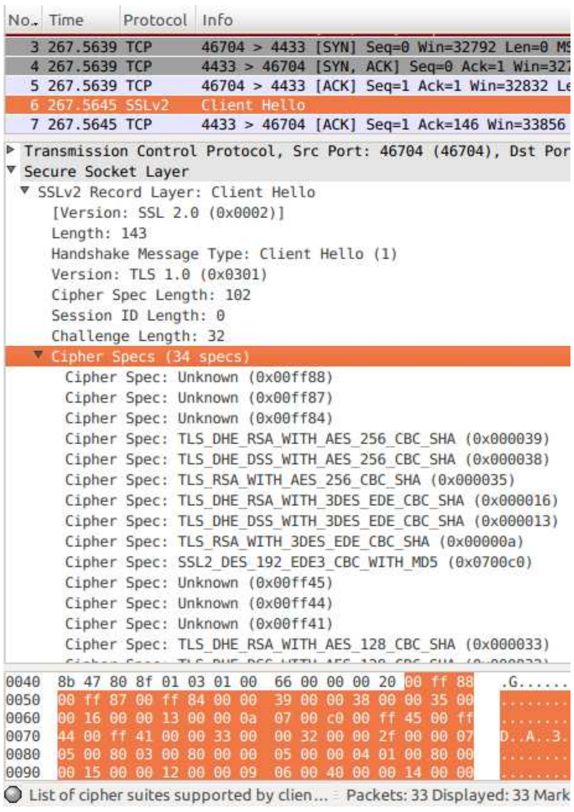
It is worth mentioning that permutation is the most expensive operation in the algorithm. According to Amdahl's law, such large improvement can be achieved only by subsequently shaving off the largest cost factors in the task [9]. Only pre-calculating the bits during permutation brings ~30% performance improvement. The performance comparison of the original IBC application (graphical application) with the IBC integrated with OpenSSL environment is presented in Table 1.

Table 1. Performance Comparison (3 GHz Processor)

Key size [bit]	**Number of rounds**	**Performance** [Mbps]	
		IBC graphical application	**IBC in OpenSSL**
128	8	2.70	63.62
192	10	2.37	52.12
320	12	2.11	44.12

6.2. SSL/TLS Connectivity

New ciphersuites containing IBC have been created in the *libssl* library. Thanks to that it is possible to use the cipher to encrypt data transmitted in SSL/TLS connections.


```
No.  Time       Protocol  Info
  3 267.5639 TCP       46704 > 4433 [SYN] Seq=0 Win=32792 Len=0 MS
  4 267.5639 TCP       4433 > 46704 [SYN, ACK] Seq=0 Ack=1 Win=327
  5 267.5639 TCP       46704 > 4433 [ACK] Seq=1 Ack=1 Win=32832 Le
  6 267.5645 SSLv2     Client Hello
  7 267.5645 TCP       4433 > 46704 [ACK] Seq=1 Ack=146 Win=33856

▶ Transmission Control Protocol, Src Port: 46704 (46704), Dst Por
▼ Secure Socket Layer
  ▼ SSLv2 Record Layer: Client Hello
      [Version: SSL 2.0 (0x0002)]
      Length: 143
      Handshake Message Type: Client Hello (1)
      Version: TLS 1.0 (0x0301)
      Cipher Spec Length: 102
      Session ID Length: 0
      Challenge Length: 32
    ▼ Cipher Specs (34 specs)
        Cipher Spec: Unknown (0x00ff88)
        Cipher Spec: Unknown (0x00ff87)
        Cipher Spec: Unknown (0x00ff84)
        Cipher Spec: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x000039)
        Cipher Spec: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x000038)
        Cipher Spec: TLS_RSA_WITH_AES_256_CBC_SHA (0x000035)
        Cipher Spec: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x000016)
        Cipher Spec: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x000013)
        Cipher Spec: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x00000a)
        Cipher Spec: SSL2_DES_192_EDE3_CBC_WITH_MD5 (0x0700c0)
        Cipher Spec: Unknown (0x00ff45)
        Cipher Spec: Unknown (0x00ff44)
        Cipher Spec: Unknown (0x00ff41)
        Cipher Spec: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x000033)

0040  8b 47 80 8f 01 03 01 00  66 00 00 00 20 00 ff 88   .G......
0050  00 ff 87 00 ff 84 00 00  39 00 00 38 00 00 35 00   ........
0060  00 16 00 00 13 00 00 0a  07 00 c0 00 ff 45 00 ff   ........
0070  44 00 ff 41 00 00 33 00  00 32 00 00 2f 00 00 07   D..A..3.
0080  05 00 80 03 00 80 00 00  05 00 00 04 01 00 80 00   ........
0090  00 15 00 00 12 00 00 09  06 00 40 00 00 14 00 00   ........

List of cipher suites supported by clien...   Packets: 33 Displayed: 33 Mark
```

Fig. 2. Client hello message.

The encryption algorithm used in a session is agreed between client and server during the handshake procedure. The client sends a Client Hello message to the server. Example of such a message is shown in Fig. 2. The Client Hello message contains the list of cipher suites supported by the client. The list is ordered by preference (usually more secure ciphersuites are more preferred). IBC ciphersuites are identified as unknown by Wireshark software because its IDs are from the private range and Wireshark does not recognize them.

The server choses the first supported ciphersuite from the list, and sends its ID to the client in a Server Hello message. It corresponds to the "DHE-RSA-INDECT320-SHA" ciphersuite, which uses the IBC cipher with a 320-bit key.

In order to successfully negotiate usage of IBC, both client and server must use a modified version of library. If one of the peers uses original SSL/TLS library (without IBC ciphersuites), then the most preferred ciphersuite which is known by both peers will be agreed (usually a ciphersuite containing the AES algorithm). This means that it is possible to establish a secure connection between peers which are using modified and original libraries.

7. Conclusion

Integration of a new cryptographic algorithm requires a number of modifications to the OpenSSL code. In particular, new SSL/TLS ciphersuites containing a new cipher must be created in order to make it available for usage in SSL/TLS connections. As an example, the integration of the IBC cipher was presented in this paper. Overall, implementation of IBC required more than 3000 changes in source code.

The cipher code was completely rewritten using low-level C language. Because of this, we were able to gain a significant performance boost over the original graphical application. Results of performance tests show above 23*x* performance improvement.

Besides performance, compatibility requirements were important during the implementation. We were able to preserve full binary and source compatibility with the original version of OpenSSL. This was proven in compatibility tests by the *abi-compliance-checker*. Because of this, it is possible to use the most convenient way of deploying IBC libraries — replacing existing OpenSSL shared library files with the modified ones.

Using the modified version of OpenSSL, it is possible to establish secure connections over the Internet using the new cipher to encrypt transmitted data. If both client and server use modified libraries, this new cipher is negotiated by default. The next step will be deploying the implemented IBC algorithm in the security architecture for Law Enforcement Agencies [10].

Acknowledgment

The implementation and evaluation of cryptographic algorithm have been financed by the European Regional Development Fund under the Innovative Economy Operational Programme, INSIGMA project no. POIG.01.01.02-00-062/09.

References

[1] OpenSSL Project. Form http://www.openssl.org

[2] RFC 6101. (2011). *The Secure Sockets Layer (SSL) Protocol Version 3.0*.

[3] RFC 5246. (2008). *The Transport Layer Security (TLS) Protocol Version 1.2*.

[4] Niemiec M., & Machowski Ł. (2012). A new symmetric block cipher based on key-dependent S-boxes. *Proceedings of ICUMT 2012*. Saint Petersburg, Russia.

[5] RFC 5932. (2010). *Camellia Cipher Suites for TLS*.

[6] Ponomarenko, A., & Rubanov, V. (2010). Automated verification of shared libraries for backward binary compatibility. *Proceedings of VALID 2010*. Nice, France.

[7] ABI compliance checker. From http://ispras.linuxbase.org/index.php/ABI_compliance_checker

[8] Niemiec, M., Dudek, J., Romański, Ł., & Święty, M. (2012). Towards hardware implementation of INDECT Block Cipher. *Proceedings of MCSS 2012*.

[9] Marowka, A. (2012). Extending amdahl's law for heterogeneous computing. *Proceedings of ISPA 2012*. Madrid, Spain.

[10] Urueña, M., Machník, P., Niemiec, M., & Stoianov, N. (2014). Security architecture for law enforcement agencies. *Multimedia Tools and Applications*.

Piotr Jurkiewicz is currently a M.Sc. student at the Department of Telecommunications at AGH University of Science and Technology, Krakow, Poland. His research interests include software defined networking, flow aware networking, fast packet processing and high performance software. He is an open source software contributor. In 2013 he was a Google summer of code student at ON.LAB.

Marcin Niemiec obtained his M.Sc. and Ph.D. degrees in telecommunications at the AGH University of Science and Technology, Krakow, Poland, in 2005 and 2011, respectively. He has also studied at the Universidad Carlos III de Madrid. He is an assistant professor at the Department of Telecommunications, AGH University of Science and Technology. His research interests focus on security and data protection, in particular security services, symmetric ciphers, cryptanalysis, malware, intrusion detection, and quantum cryptography. He is the co-organizer of a number of international meetings, workshops, and conferences. He has actively participated in European Commission’s 6th and 7th Framework Programmes (ePhoton/ONE+, BONE, SmoothIT, INDECT), Eureka-Celtic (DESYME), and several national projects. He is the recipient of the Best Paper Award from IEEE GLOBECOM 2012. He has co-authored over 60 publications and reports (papers, deliverables, book reviews, IETF drafts, and books.