Academic Year 2002/2003

Warsaw University of Technology
Faculty of Electronics and Information Technology
Electrical and Computer Engineering

Bachelor of Science Thesis

Names of Students: *Robert Jastrzębski, Michał Poręba*
Title of Thesis: *E-forms with Digital Signature*

Supervisor
Rajmund Kożuszek, M.Sc.

Evaluation: ……..
……
Signature of the Head
of Examination Committee

Electrical and Computer Engineering

Name:	*Robert Jastrzębski*
Date of Birth:	*September 29, 1980*
Starting date of Studies:	*October 1, 1999*

Curriculum Vitae:

I was born on September 29, 1980 in Warsaw, Poland. I attended XXVIII LO under the patronage of Jan Kochanowski in Warsaw. Currently, I am a student at Warsaw University of Technology, at the Faculty of Electronics and Information Technology. My undergraduate specialization is Computer Systems and Networks. I am an owner of a FCE English certificate and ZD German certificate.

……

Signature of the Student

B.Sc. Examination

Examination was held on …….

With the result: …….

Final Result of the Studies: ……..

Suggestions and Remarks of the B.Sc. Examination Committee ……

……...

……...

Electrical and Computer Engineering

Name:	*Michał Poręba*
Date of Birth:	*September 25, 1980*
Starting date of Studies:	*October 1, 1999*

Curriculum Vitae:

I was born on September 25, 1980 in Warsaw, Poland. In 1999, I graduated with honors from LO under the patronage of T. Kościuszko in Pruszków. I am studying at the Warsaw University of Technology in the Faculty of Electronics and Information Technology, specialization in Computer Systems and Networks. In 2000, I passed the examination for the Certificate of Proficiency in English. I am interested in science fiction, role-playing games and kendo.

……

Signature of the Student

B.Sc. Examination

Examination was held on …….

With the result: …….

Final Result of the Studies: ……..

Suggestions and Remarks of the B.Sc. Examination Committee ……

……...

……...

SUMMARY

The thesis presents the analysis of the techniques that are available to create a system for creating and managing digitally signed e-forms. Various document formats, submission and processing methods are described and compared in the perspective of using them in today's Polish reality in order to construct a system that would allow an ordinary citizen to fill certain forms (e.g. tax form, driver license application etc.), sign them electronically to ensure the data integrity and users identity, and submit said forms to the local government office.

Basing on the result of this analysis, a sample system of such a kind is implemented and described in more detail in the thesis. The system uses a combination of well-known technologies like PHP, ActiveX, and C++ language; and is based on Apache web server, MySQL database engine and MS Internet Explorer web browser. Such a system is universal and extensible and can be tuned up to suit the needs of various local governmental offices and other institutions that would be interested in using the electronic signature in today's society.

Keywords: e-forms, digital signature, electronic signature

Table of Contents

I. Introduction *(Michał Poręba)*

1. E-forms: Advantages and Risks

During the last two decades, the Internet has evolved from a small-scale scientific and military experiment to the level of one of the most important inter-human communication media. No longer is it a read-only medium, a means of presentation only: more and more interactive services are being introduced every day. Business usage of the Internet is a fact for almost a decade now, and national governments also use the global network for the support of their activities and interaction with the society. The situation evolves quickly: every day more and more information is submitted in the digital form instead of creating or submitting them the traditional way. E-forms – digital interactive forms that are filled by the user – are a large part of this information interchange.

Thanks to the introduction of the digital form of documents, their transport and processing time is shortened significantly, their generation, storage, and processing costs can be lowered dramatically. Archiving, modification and publishing of documents in electronic form is much easier than when using traditional (i.e. paper) format. Electronic document databases give more opportunities for document processing and analyzing, including such trivial examples as searching for the proper document or finding a set of documents basing on a given criteria.

However, there are also some risks involved in electronic handling of the documents. Like their paper counterparts, digital messages can be a target of forgery attempts, data manipulation etc. It is definitely easier to change some data in a document in a digital form than on a paper document, and doing it without leaving any traces of the manipulation at all is not so difficult, either. Therefore, some way of ensuring the document integrity in its digital form is required, along with some way of checking the author's identity.

2. Digital Signature: Reducing the Risk

In order to meet these requirements, a digital signature concept has been introduced. This concept consists of methods of advanced cryptography that are used to generate and encrypt a digital 'digest' of the document and relate it to its author – thus making it a fully-operational 'signature', and appropriate physical infrastructure (so-called PKI – Public Key Infrastructure) that allows any interested party to identify and verify the signature of any other user of the system. In general such a system is based on an asymmetric key concept: a user has a pair of keys at his disposal, one of them being called 'private', and the other – 'public'. Private key is known only to its owner and is safely stored in a smart card (or, more rarely, on a computer disk), while the public key is publicly available. A message signed with the person's private key can be verified only by the usage of this person's public key. If the decoding process returns the 'invalid' result, it means that either the message had been compromised and changed between being signed and decrypted, or the keys are not a valid pair – that is, the message have been signed with a key that does not belong to the same person as the public key.

The infrastructure and signature generation methods can span a single company or organization, can be accepted as a means of communication between some bodies of different kind (for example for the purpose of money transactions between select banks that agreed to some method of digital signing), or can be regulated by the legal

system of a given country to define a single standard of digital signature that is legally equivalent to the personal handwritten signature in its aspect of the legal statement of will. Day by day, new countries all over the world introduce legal bills regulating this method of data interchange and authentication: for example in 1997 such a bill was introduced in Germany, in 2000 a Digital Signature Act was signed by the President of the USA, and in 2001 a similar bill was signed by the President of Poland.

3. 'E-forms with Digital Signature': Aim and Structure of the Thesis

In this thesis, we are to analyze the technical possibilities and methods of introducing digitally- and electronically-signed[1] e-forms in today's Poland, when it comes to their usage by an average citizen of the country. While certain companies and organizations already use the digital signature for quite a significant time (e.g. KIR S.A. – National Clearing House responsible for inter-banking money transfers has such a system operating since 1991), similar solutions for open public are still scarce. In the thesis, we investigate the technologies that could be used for such a purpose, along with their benefits and fallbacks, basing on an example of a system of submitting e-form applications to the local governmental agency. Eventually, we implement a sample system that could work on a powiat- or voyevodship-level and allow the citizens to submit electronically signed e-forms to the local government office.

First part of the thesis consists of a preliminary analysis of the requirements that would stand before a system of such a kind, simultaneously outlining the basic stages of a process of creating and signing an e-form. Second part contains a more detailed analysis and description of more popular of the currently available technologies that can be used for the purpose of e-forms generation and digital signing. Finally, third part of the document contains a specification of the sample system we decided to implement, along with implementation notes and remarks on more interesting aspects of the task. As a conclusion, a few methods of expanding the system are presented.

[1] The difference between *digital* and *electronic* signature is basically a legal one: an electronic signature is a digital signature that adheres to the legal norms of Polish law.

II. Preliminary Specification *(Michał Poręba)*

1. Project Assumptions and Constraints

According to the Electronic Signature Bill, all governmental agencies and offices must be prepared to manage electronically-submitted, digitally-signed official documents as soon as the year 2004. This thesis shall investigate the possibilities of creating and maintaining a client-server system allowing an average citizen of Poland to contact his local governmental office and submit digital forms such as driving license application, tax form and similar documents.
Due to this assumption, the system should be able to interact with users of different hardware and software platforms (PC, Mac, Windows™, Linux…). It should not require nor promote any commercial document format, submitting method etc.
Adding new forms and modifying existing ones should be allowed.

2. Functional Requirements

Hardware and Software Platform

In current Polish reality the most popular client system is a PC-class computer (Intel or AMD x86-family processor) equipped with Windows™-family operating system, Internet Explorer™ web browser and a low or medium-bandwidth Internet link. The system should be primarily designed to work in such a configuration.
Additionally, there should be a possibility of submitting the e-forms from different platforms: most notably Linux or Unix family and MacOS.

Document Creation

Document contents and layout should be easy to comprehend, presented in a clear, organized and easily readable way.
Document should have the same or very similar appearance when viewed using different hardware and software.
User assistance should be included – in the form of context help, tips, wizards, or in similar manner.
Form management should be intuitive and as simple as possible.
User-entered data should be verified and its correctness assessed before the document is signed.
All the data entered must be secure from any external hazards; any manipulation by a third party shall be prevented.

Data Verification

User-entered code should be checked for presence of malicious or illegal contents (SQL commands, some special characters, letters entered instead of digits etc.).
Data consistency and cross-validity should be checked (e.g. PESEL versus user-entered gender, age etc.). An option to override the mechanism in specific situation should be introduced.

Document Signing

Signing process should adhere to the legal norms (the Electronic Signature Bill).
The system should work with various certified signing equipment.

Document signing process should be controlled by the user – the signing cannot be done automatically, it must be initiated by the user as an 'act of will' as the legal norms require it.

Document Submitting

Document submitting process should be controlled by the user.
Submitting through the Internet should be done using a secure encoded communication channel.
The user must be sure his data is submitted to the second-party it is intended to.
If the user has no access to digital signing equipment, he should be able to print out the form and submit it via traditional mail.

Alternative Document Acquisition

If the user doesn't apply digital signature and the document is printed and submitted via traditional mail instead, the printing format must be independent of the hardware and software platform used – the document is to be scanned and OCR-ed by the office.

Signature Verification

The digital signature must be verified according to the legal regulations, by contacting the certified Center of Authorization. The document integrity and author identity must be checked.

Data Verification Repeated

Data verification process must be repeated server-side in order to prevent any changes introduced after primary verification and before signing – to prevent both unintentional mistakes and hacking attempts.

Data Management

All the data shall be stored in such a way that its manipulation and processing is relatively easy. This means for example inserting the critical data into some kind of relational database or, if such a need arises, using a full-text database.

Archiving

The documents are to be stored for a long period, as the legal regulations require.
Compatibility problems with reading the document in the future should be minimized (i.e. a document format that is not likely to become obsolete soon is required).
Additional data regarding the key, certificate, date of signing etc. should be attached to the archived document if required by the office.
Finding the specific form basing on various search criteria should be possible in a realistic time.

System Maintenance

New forms with validation rules should be easy to add.
A possibility of modifying and/or removing existing forms should be present. This includes modification/adding/removing of any or all of validation rules, of form fields or both.
Help system should be easily adjustable.
Some software for writing the forms for the system may be prepared.

III. Analysis

This part is devoted to the analysis of possible methods of implementing a digitally signed e-form management system in the light of the requirements specified in the previous chapter.
An analysis of the most popular document formats that could be used in the project is performed, then comes the filling and submitting methods analysis and their comparison, along with summary on both formats and methods. Next part contains the description of encryption algorithms used in digital signing process, and investigation on various crypt libraries that can be used with different operating systems. Finally, the Bill on Electronic Signature is described in order to specify the legal requirements that would stand before the project.

1. Popular Document Formats *(Michał Poręba)*

1.1 RTF

RTF (Rich Text File) is a Microsoft-designed document format that was introduced to allow data interchange between various devices, operating environments and operating systems.

RTF file contents

An RTF file consists of unformatted text, control words, control symbols, and groups. Reading the document requires separating the control information from plain (unformatted) text, acting upon control information and then inserting appropriate text in appropriate places (called 'destinations').
A **control word** is a special command that marks printer codes or application instructions inside the document body. It begins with backslash and takes the following form: *LetterSequence*<*delimiter*>*. The delimiter may be space (if so, it is treated as a part of the control word and is not displayed in the document), a non-letter, non-digit character (treated not as the part of the control word) or a digit sequence, possibly preceded by a minus sign (this entire number is treated as a parameter to the control word and must be delimited by other means at the end). A control word is generally used to set some property on, off or assign it some value. It may also be used to select a destination (e.g. *footnote*) for the selected group.
A **control symbol** begins with backslash and consists of one character treated as special, non-delimited. Control symbols are mostly used for inserting special characters into the document body.
A **group** is a sequence of characters or other data delimited by the braces: '{' at the beginning, and '}' at the end. A group state (control information attributed to a group) specifies the status of an entire block that is included inside its body – this typically means character or paragraph properties (align, facing etc.), or table or section properties (number of rows, number of columns etc.).

RTF document structure

Any RTF document consists of two parts: header and document area.
The **header** contains information such as RTF version, character set and fonts used – along with the possibility of embedding a font definition inside the document header. It may also contain: information on the code pages used, a file table (only if the document is composed from subdocuments), a color table (defining the colors used in

the document), a style sheet (containing a set of styles that are used in the document), a list table (e.g. later versions of MS Word™ use header to store information about lists rather than the paragraphs in the document body themselves) and a revision table (used to keep track on the document revisions and versions).
The **document area** stores the proper document contents. It may contain an information block describing the document author, title, subject, creation time and similar properties. Then there can be placed a document-formatting block with information regarding the document appearance, margins, details about spacing, footnotes, and other document-related technical info. After these sections, a number of one or more text sections follow. These sections contain the 'proper' document text, data, tables, pictures etc., along with information on it's formatting.

Summary

The current RTF specification (version 1.6 established in A.D. 1999)[1] does not support any internal digital signature mechanisms at all. It also lacks any technology that could be used in a sensible way to present and allow for filling an e-form. Of course, it can be signed 'externally', as every other document format, but this leads to unwanted and unnecessary complications. In addition, the popularity of RTF seems to be currently falling because of the growing popularity of PDF format in technical usage and *.doc* in business.

1.2 MS Word *.doc*

This is the primary document format of the Microsoft Word™-family word processors. Due to the large number of the users of this software, during the last few years the *.doc* format, although unwieldy and large, became one of the most popular means of data interchange especially between business agencies and unqualified computers users.
Word document is a *docfile* – an OLE-conformant[2] binary file that contains streams of data stored as linked lists of file blocks. They are (and must) be accessed by using an appropriate OLE APIs.
A Word file consists of a main stream, a summary information stream, a table stream, a data stream and a number (zero or more) of individual object streams that contain private data for the embedded objects. These objects may include anything from pictures to sounds to spreadsheets and are not interpreted by the Word itself – appropriate OLE APIs are used to access them. The main document stream consists of Word file header, the text, and the formatting information. If the file has been saved incrementally (without erasing the previous values, this is a default setting for the Word auto-save feature), additional content is appended at the end of the first stream.

Summary

The Word-document format is subject to change every time a new version of MS Office™ is issued. Owing to this, the compatibility issue can be a serious problem: while in newer versions of Word some methods of digital signing can be introduced, older versions of the software won't support them. Moreover, a docfile does not contain any internal mechanisms for e-forms creation; and finally, MS Office™ is a solution fitting for business companies rather than for home use.

[1] For more information on RTF document format, see [8]
[2] OLE – Object Linking and Embedding – a technology introduced by Microsoft in order to allow for linking and inclusion between various document types.

1.3 PDF

The Adobe PDF (Portable Document Format) in version 1.5 is probably the most popular document format when it comes to the exchange of technical data between the users of various application software, hardware, and operating systems, especially over the Internet. Originating from the PostScript language files, this file format is also quite popular in government and business usage when it comes to publishing the data on the web.

PDF file contents

A PDF file contains a PDF document (see below) with an addition of data regarding its structural information, all encapsulated as a single self-contained sequence of bytes.

A PDF document is based on describing individual pages, which are described by a content stream that contains a sequence of graphic objects to be printed to the output device. The objects are not to be read in a sequence – a viewer application processes the file by following the references from object to object. When the document is modified, the previous data is not overwritten – rather, the new object is added as a new section and all the relevant links are updated. All the elements are indexed in the cross-reference table that is placed at the end of the file.

As an addition to static data, these objects may describe some active or interactive objects that are available only in electronic form of the document: this may range from internal and external hyperlinks to sounds and animation to file attachments. A document can also have added triggers that react to user actions (e.g. mouse movements) and undertake specific actions, and forms that can export the data entered by user to other applications. An object can also contain some higher-level application data that describes its logical structure and may be used for information interchange with other applications, for searching and editing parts of the document etc.

The document contents may be optimized for faster viewing or searching using a *Linearized PDF* file organization. In order to reduce the file size, the PDF format supports many methods of internal compression for both graphics and text, including JPEG, and LZW (Lempel-Ziv-Welch) among the others. PDF also allows for font embedding, including the possibility of embedding only such a subset of the font definitions set that is really used in the document. There is also a set of 14 fundamental fonts that is required to be implemented in every PDF-reading software/device, therefore making it's embedding not needed.

PDF document structure

Characters in PDF file are divided into three groups: regular, delimiter and white-spaces. The following characters are treated as a white-space: null, tab, line feed, form feed, carriage return and space. Delimiters are all four kinds of braces, slash and per-cent sign – they are used to delimit syntactic entities such as strings, arrays, names, and comments. All other characters are treated as regular. A sequence of regular characters is called a token. One or more tokens make up an object. Objects may be of one of the eight types: Boolean value, integer/real value, string, name (an unique, atomic identifier), array (a sequential set of different objects), dictionary, stream, and null.

These objects are arranged into the PDF file the following way: first comes a one-line file header specifying the version of PDF specification the document conforms to. Then comes the document body (containing all the document data, as described above), the cross-reference table and, finally, a trailer giving the location of the cross-

reference table and of certain special objects within the file body. Every time the incremental updates are made to the file, a new part of the document body, a new cross-reference table, and a new trailer is added at the end of this structure.

PDF forms

An interactive form used in PDF format, is a collection of fields used for gathering information interactively from the user. A PDF document may contain any number of fields appearing on any combination of pages, all of which make up a single form spanning the entire document. These fields may include: buttons, checkboxes, radio buttons, text fields, choice fields. Digital signatures are also included here.
Arbitrary subsets of this document-spanning form can be imported or exported from the document. This can involve sending the data to the specified server in the specified format – including HTML, XML and the Adobe-specified FDF (Forms Data Format). The data may be processed server-side and sent back to the source document, triggering some actions.

Summary

The PDF format offers many inherent methods of digital signing and encrypting the document. Thanks to its object structure and the presence of the cross-reference table, digital signatures can be applied to the document in series, and previously signed versions of the document can be accessed and their consistency verified even after the document had been modified a few times by different persons. As described above, the Adobe format has also internal support for e-forms creation and filling. When it comes to comparing the self-contained file formats, Acrobat's PDF is definitely the most well suited and prepared to handle e-forms with digital signatures.[3]

1.4 XML

XML (Extensible Markup Language) is a successor to SGML (Standard Generalized Markup Language), optimized for usage as a data exchange format in the Internet environment and for document parsing simplicity. This language (or a meta-language, because it is used to formulate other markup languages) concentrates on organizing and describing the document contents and logical structure rather than its appearance – the document appearance can be interpreted by the reader program and/or defined by the usage of external style sheets.[4]
An XML format is getting increasingly popular these days. MS Office™ 2003 is supposed to be based on an XML format, as well as some of the databases and many specialized applications. A digital signature format definition in XML was officially released in spring 2003, and HTML language had been reformulated in XML as XHTML.

XML markup

An XML document has a tree-like structure composed of elements, with boundaries marked either by start-tags and end-tags, or – in case of an empty element – by an empty-element-tag. A starting tag is marked by triangular brackets (<*element_name*>), an ending tag is of the form <*/element_name*>, and empty element is of the form <*element_name* />. As seen in these examples, an element name is stored in the tag. Along with it, some attributes may be defined in a strictly regulated way, for example: <*car type="truck" color="red"* />.

[3] For more info on PDF document format, see [7]
[4] For XML format specification and additional information, see [2]

A non-empty element can contain other elements to create the mentioned hierarchical structure. For example, a simple XML document may look as follows:

```
<?xml version="1.0" encoding="Windows-1250"?>
<garage owner=”Michael”>
<car type=”truck”>
<driver name=”John” />
<cargo>fresh food</cargo>
</car>
<car type=”taxi” color=”blue”>
<driver name=”Hans” />
</car>
</garage>
```

Names of the elements used and allowed tree structure are defined by the creator of the document or application in which the document is going to be used. Basic way of defining and describing a new document type that will be used in an application is preparing a Document Type Definition (DTD). Such a definition describes in a formal way all the elements that can be used in the document, their attributes (allowed or required, along with possible default values), and contents (other elements with their number possible, character sequence etc.).
A DTD for the file above could look as follows:

```
<!ELEMENT garage (car)* >
<!ATTLIST garage
      owner CDATA #REQUIRED
      >
<!ELEMENT car (driver?, cargo?) >
<!ATTLIST car
      type CDATA #REQUIRED
      color CDATA
      >
<!ELEMENT driver EMPTY >
<!ATTLIST driver
      name CDATA #REQUIRED
      >
<!ELEMENT cargo (#PCDATA) >
```

Summary

Although there exist a few ways of using XML for data presentation, this format is destined mostly for data storage, interchange, and processing. It can contain other documents or links to such objects, and has a defined and widely recognized digital signature format. A natural – and compatible with most of the client software – way of presenting XML-stored data in the web environment would be an XHTML document format. XHTML can be also used for designing an e-form to collect required data, and for passing it to the server to form an XML document. XHTML is described in more detail the next chapter.

1.5 XHTML[5]

XHTML (Extensible Hypertext Markup Language), essentially an HTML 4.01[6] updated to conform to XML 1.0 standards, is the basic language used for the construction of web pages. A predefined set of tags that are allowed to be used in XHTML is defined in three basic DTDs that are publicly available on the World Wide Web Consortium web pages (http://www.w3.org/TR/xhtml1/#dtds). These elements are supported by a vast majority of modern web browsers (technically named *user-agents*) and can be used to describe the appearance of the page on the screen or other output device. Web browser producers often create their own extensions to the HTML, but these are not always supported by other user-agents, while the W3 documents are treated as a standard every program should adhere to.

XHTML forms and submitting

XHTML supports a defined set of elements that are used to build an on-line document form. These include text fields (with both and 'password-like' method of displaying entered text), text areas, drop-down menus, radio buttons and submit buttons. Data entered into the form can be submitted to the target page/script/program on the server in an HTTP request using one of the two methods: GET or POST. The GET method is based on appending the data to the requested document URI (Uniform Resource Identifier), while in the POST method, the data is sent inside the message body. Former of the methods is supposed to be used in simple tasks such as querying a web search engine, while the latter is to be used when there is some data manipulation or other visible effect expected as a result of analyzing the input data (for example, making a money transaction with the bank, or ordering a pizza).
Any data that arrives from the user through the web server can be processed server-side by a specified script or program and given actions may be undertaken basing on its contents. This usually means at least displaying another web page (for example as in search engines), and may also include updating internal database, sending an e-mail or Usenet news article etc.

Summary

When it comes to designing a web application using a web browser, XHTML is the only possible choice, that can be augmented with a great many of supporting technologies such as JavaScript (to form a so-called *Dynamic HTML*), Macromedia Flash™ etc. As described above, this language supports form creation very well, and although it cannot be self-signed as the PDF format can, it is small and flexible enough to be included into XML file for signing and data storage.

2. Document Generation Methods *(Michał Poręba)*

2.1 Basic Methods Available

Offline document passing

The form is prepared as a document in one of the well-known commercial standards such as .pdf, .doc or other, using one of the commercial-grade word processors. The document is then sent to the user by means of e-mail, or downloaded by the user from ftp server or web page using appropriate software and protocols. The user then fills

[5] For full XHTML format specification, see [9]
[6] For information and formal specification of HTML, see [3]

the form using appropriate software and signs the document, possibly by appropriate plugin integrated with the user's word processing software. Then the signed document can be sent to the office via e-mail (digital signature could also be done in this stage, by signing an entire e-mail message) or any other appropriate method. This can be broken up into the following stages:

- client:
- downloading the proper e-form document
- filling the e-form
- assessing data validity client-side
- signing the document
- submitting the document
- server:
- receiving the document
- verifying digital signature
- assessing data validity server-side
- issuing confirmation to the user
- managing received data

Basing on the results of analysis in the previous chapter, an Adobe PDF format would be preferred if this method was to be used.

Online form filling

This method makes use of web browser and HTTP server as a means of communication between client and server systems to generate the form on-line. The form is created in a form of a web page (using XHTML with possible addition of other appropriate techniques), and server-side scripting (PHP, JSP, ASP etc) is used to manage the data on the server side. The document is signed by using an appropriate web browser plugin that is to be installed on the client's machine and submitted using the HTTP protocol again[7]. Java Script can be added to the pages in order to increase the client-side functionality. The whole process would contain the following stages:

- selecting and downloading the proper e-form document (client/server)
- filling the e-form (client)
- submitting the form (client/server)
- assessing data validity server-side (server)
- generating the document to be signed, containing previously entered data (server)
- retrieving the document (server/client)
- signing the document (client)
- submitting the document (client/server)
- verifying digital signature (server)
- comparing the data received with previously validated one (server)
- issuing confirmation to the user (server/client)
- managing received data (server)

Dedicated client

In this method, a special program is to be written and installed on the client's machine, similarly to the 'Program Płatnika' of Polish ZUS. It contacts the server directly and is responsible for all the operations done to the document. The program should be easy to update – both the main engine and e-form templates.

[7] For detailed info on the HTTP protocol, see [4] and [5]

- client:
- selecting the proper e-form document
- filling the e-form
- assessing data validity client-side
- signing the document
- submitting the document
- server:
- receiving the document
- verifying digital signature
- assessing data validity server-side
- issuing confirmation to the user
- managing received data

2.2 Methods Comparison

The following table compares the most important characteristics of the highlighted implementation techniques.

Phase	Aspect	Method		
		Offline passing	Online	Dedicated client
General	Portability w/r to different hardware and software platforms	Poor (specific commercial-grade software for specific platform is required)	Excellent (any system equipped with web browser can be used)	None (specific client for different operating systems is to be prepared)
	User costs	Very High	Low	Very Low
Document Generation	Presentation clarity	Very Good	Good	Excellent
	Document appearance identity under various platforms	Very Good	Good (dependent on the web browser – appropriate page design is needed to minimize differences between various user agents)	Excellent

Phase	Aspect	Method		
		Offline passing	Online	Dedicated client
	User assistance during the document generation phase	Poor (some macros and tips can be introduced as a part of the form document)	Very Good (on-line hints, context help as new or pop-up web pages can be added, can be easily modified on the server)	Good (built into the client program)
	Server and network usage during document generation phase	No	Yes	No (or Yes, depending on implementation)
Primary Document Verification	Time of preliminary document verification	Very Short	Short to Very Long if the server is overloaded	Very Short
	System security against bad data	Poor (validation must be done by macros)	Very Good (enforced by the server)	Very Good (hard-coded into the program)
	Data security	Very Good	Good (data transmission must be secured)	Excellent
Document Signing	Integration with signing system	Medium (appropriate plugins are to be used)	Medium (appropriate plugins are to be used)	Excellent (signing can be performed by the document editing program)
Document Submitting	Data transmission security	Depending on submission method	Very Good (if using encoded transmission)	Good (own protocol and encoding can be used)
	Data extraction	Difficult to Very Difficult	Easy	Very Easy
	Similarity of printed document vs. OCR template	Very Good (may differ because of printer settings)	Poor (may differ because of different browsers and printer settings)	Very Good (requires own handling of printing)

Phase	Aspect	Method		
		Offline passing	Online	Dedicated client
Document Archiving	Compatibility prospects	Medium (commercial standards change)	Very Good (W3 standards)	Poor (own 'standard' defined)
	Possibilities of including additional data	Poor	Very Good	Very Good
System Maintenance	Modification possibilities	Very Good (e-form documents can be modified on the server before download)	Good (e-forms, help system and verification rules can be adjusted on the server)	Poor (software must be changed every time the changes are introduced)
	Version control	Poor (previously-downloaded files can be used instead of the newest ones)	Very Good (new documents are downloaded from the server every time)	Medium (using of the most current software version may be enforced)

2.3 Summary

Each of the methods outlined above can be implemented in a large number of different ways and techniques, giving a great variety of methods that can be employed for submitting an electronically signed e-form. For the system that is to be used by average people – not by commercial companies – offline document passing method carries a set of main disadvantages that disqualify its usage in current situation. While the system would be relatively simple to implement on the user and transportation sides, it enforces the usage of specific software from a selected commercial company (in some cases demanding a specific operating system, and always promoting the company in question), which should be obviously avoided. Second, this software is rarely affordable for individuals. Although this reason doesn't refer to the Adobe software, which seems to be the leader when it comes to the offline document passing method, there is no guarantee that the company's policy won't change in near future and Acrobat Reader™ won't be given away freely anymore, or that it will have the functionality to fill and submit form data. Third, commercial standards are subject to change without warning, potentially causing many problems in the archiving process and later data retrieval.

The choice between online method and the usage of a dedicated client is not as obvious, as the differences between them in some areas can be not so obvious either. Online method requires the web browser plugin, which can be perceived as a very limited client program. On the other hand, a 'large' client could be augmented by publishing the forms and validation rules online and allowing it to contact the server to synchronize the rules and form versions. The plugin must be prepared separately for all the systems supported, as the client program. The advantage of online method

is that even if there is no plugin for signing the document under the specific (maybe very exotic) platform, the filled form can still be printed and submitted via traditional mail. Also, downloading, installing and upgrading the plugin can in many cases be done automatically by using the browser built-in mechanisms, saving time and work of the user. Additionally, there had been much controversy about the 'Program Płatnika' (that is a part of the 'dedicated client' method) in Poland lately, which leads to a whole new set of yet-untouched social and legal problems, like: who should be the owner of the client software; whether the code and transmission protocol should be kept secret (to increase safety against crackers) or freely-available (to allow creation of independent client versions by third parties); and many more. Designing the system as a web application – that is a common sight for most of the web users – would allow for avoiding these problems and minimize the possibility of creating a negative atmosphere around such a new 'product' as the electronic signature.

3. Digital Signature Algorithms *(Robert Jastrzębski)*

3.1 The overview

The abbreviation of RSA is Rivest, Shamir and Adleman - from first letters of the names of its authors. It is a public key cryptosystem used for encryption and authentication, which was invented in 1977. The RSA algorithm will be used in the project, because it provides high security combined with average computation speed and ease of implementation. Verification of authenticity is an important point of the coding phase of the project, and the reason is being shown below:

Let us imagine a situation when one person wants to send a document to another person. The main point is that the receiver should have an opportunity to verify if the document is authentic - if it has been received with no changes applied to it on the way. To achieve this, the sender encrypts the data with the public key of the receiver, sends it, and the receiver decrypts the message using his private key.

In fact, the RSA algorithm uses two pairs of keys, not one as in standard encryption methods. The public key pair is {N,E} and the private key pair is {N,D}. The only restriction is that the size of data should not exceed the size of key N. For quite secure encryption, it is enough to keep the size of key N equal to 128 bits, for very secure encryption equal to 512 bits. Usually the data to be encrypted is the hash of the document we want to encode. Typical hashing functions are MD4 and MD5 functions, which produce 128-bit output; and SHA-1 function, which produces 160 bit output.

The situation that occurs in the project is in fact different to the one presented above. To determine the authenticity of the message the certificate owner will encrypt the document using his private key pair, and it will be checked using his public key pair from the server side. This operation will assure that the encrypted message is authentic.

However, this forces some modifications of notation, to avoid confusion in next chapters explaining the RSA algorithm. {N,E} will be called the private key pair and {N,D} will be believed to be the public key pair. Since the encryption and decryption process can be used in both ways (namely, one can encrypt using public key pair and decrypt using private key pair and vice versa, of course obtaining different results after encryption but the same after decryption), notification does not make much of a problem here.

The hashing function

The hashing function used in the project is the MD5 hashing function. The reasons why it was chosen are similar to those about the RSA algorithm. The function is very efficient together with very fast computation speed. Since the main point of the project is the implementation of the coding algorithm used for signing the document, thanks to the MIT Laboratory for Computer Science and RSA Data Security, basic functions used by the MD5 algorithm could have been downloaded and used without any restrictions. However, the full implementation of the MD5 function is not that simple, and requires some explanation.
The MD5 function takes as input a message of arbitrary length (in our case content of a file) and produces as output a 128-bit "message digest" of the input. It is strongly believed, that it is impossible to produce two messages having the same message digest, or to produce a message having a given target message digest. That is why the MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private key under a public-key cryptosystem such as RSA. The feature that it is not possible to find the message having its digest is the most important fact from the point of view of security, which is very important for the project.

The algorithm consists of five major steps listed below:

1. Append padding bits
2. Append length
3. Initialize MD buffer
4. Process message in 16-word blocks
5. Produce output

The steps will be briefly described, to show the general idea of the message digest algorithm:

1. The message is extended so that its bit-length is congruent to 448, modulo 512. It means that the message will be 64 bits shy of being a multiple of 512. Padding is performed as follows: a single "1" bit is appended to the message, then "0" bits are appended to achieve the desired bit-length of the message.
2. A 64-bit representation of the message is appended to the resulting message. Since the previously calculated message was 448 bits long, new one will have 512 bits in length.
3. A four-word buffer (A,B,C,D) is used to compute the message digest. Each one of A, B, C and D buffers is a 32-bit register. They are initialized to some fixed hexadecimal values.
4. This step is the major calculation phase of the MD5 algorithm. At first, four functions (F,G,H,I) that take as input three 32-bit words and produce as output one 32-bit word are defined. These functions will perform certain bit operations and as a result of these operations, new values of A,B,C and D will be calculated, which will be used to output the final result.
5. The message digest produced as output begins with the low-order byte of A, and ends with the high-order byte of D.

Of course, the key step of the message digest algorithm is step no.4, where bit operations are being performed. It is believed that it is impossible to reverse this process and that is why the MD5 algorithm is said to be secure.

3.2 The algorithm

Key generation

The RSA algorithm uses two pairs of keys, not one as in simple coding. In fact, these pairs of keys involve only three, not four keys. The private key pair, used for encryption of data, is {N,E}. The public one, used for data decryption, is {N,D}. It is obvious, that there must be some relationship among these three keys, to assure correct encryption and decryption of data. The dependencies among the keys result from the key generation phase, which consists of five steps listed below:

1. Generate two prime numbers *p* and *q*
2. Let $N = p*q$
3. Let $m = (p-1)*(q-1)$
4. Find *E*, coprime to m $\Leftrightarrow$ $GCD(E,m) = 1$
5. Find *D*, such that $(D*E) \% m = 1$

As steps 1 to 3 are quite clear, last two steps require a short explanation. GCD stands for greatest common divisor. Such *E* must be picked, that the equation $GCD(E,m) = 1$ would be satisfied. Left side of the equation from step five represents the *modulo* operation. Such *D* must be chosen that the reminder of the quotient would be equal to one. This equation can be transformed into such form:

$$(D*E) \% m = 1 \Leftrightarrow (D*E = k*m + 1) \wedge (k \in \{0,1,2,...\})$$

From the five steps of key generation phase also follows that:

from 1. Since *p* and *q* are prime numbers, they are odd
from 2. Since *N* is a product of two odd numbers, it is also odd
from 3. Since *m* is a product of two even numbers it is even
from 4. Since *m* is even, than *E* must be odd, otherwise $GCD(even,even) \geq 2$
from 5. Since $k*m$ is even for any *k*, and $k*m + 1$ is always odd, than since *E* is odd, *D* must be also odd

The final conclusion is that all three keys are odd numbers. The RSA algorithm will surely not work for keys of which not all are odd, nor will it work for keys, which do not satisfy the steps of the key generation phase.

Data encryption and decryption

During encryption of data, a certain "message" will be encoded. To assure that this process is performed correctly certain rule must be obeyed. Namely, size of this data must not exceed the size of key N. Otherwise the result might be improper. Of course, the same rule applies to the decryption process. Since the encryption process uses the private key pair {N,E} and the decryption process uses the public key pair {N,D}, each algorithm must involve only its own key pair.
Suppose that we want to encode a message *A*, and as a result we obtain an encrypted message *C*. The formula to perform this operation is as follows:

$$C = A^E \% N$$

To decrypt the encrypted message *C* we must perform a similar operation, resulting in calculation of message *B*:

$B = C^D \% N$

Of course, a correctly implemented RSA algorithm guarantees the identity of the initial message *A* and final message *B*. From the form of both equations a deduction can be made, that during the implementation phase a single crypto-function can be constructed. The equations differ very slightly and this fact is very convenient for the programmer.[8]

4. Crypt Libraries in Operating Systems *(Robert Jastrzębski)*

4.1 Cryptographic services under Microsoft Windows OS

CryptoAPI is a technology supported by the Microsoft Windows operating system. It provides a way for applications to get cryptographic services, like for example encryption and decryption of data. This technology supports special modules, called Cryptographic Service Providers, or shortly CSPs, which actually support all necessary cryptographic functions. They perform cryptographic operations, generate and store private keys.
To generalize the Cryptographic Service Providers, they can be divided into three major categories:

1. Hardware based
2. Software based
3. Combined - partially hardware and software based

Because private keys and cryptographic operations are isolated from the operating system in hardware based CSPs, we may say that such convention is more secure than the software based cryptography. However, using hardware based cryptographic functions may have its disadvantages. First of all, storage space of such hardware is limited, so is the calculation power. It may take longer time for hardware based CSPs to generate keys. Hardware based CSPs are usually used when extreme security is required, for example while logging with use of smart cards. Combined Cryptographic Service Providers usually involve some hardware unit (like smart card verifier), together with fast computation speed achieved by use of some software.
It is worth mentioning, that all CSPs, which are intended to be used under Microsoft Windows operating system, need to obtain a certificate from Microsoft, otherwise they will simply not work under this system.
The following Cryptographic Service Providers are being supported under Microsoft Windows operating system:

- Microsoft Base Cryptographic Provider - provides basic cryptographic functions, uses RSA technology, is not subject to United States government cryptography export restrictions and therefore can be exported to other countries

[8] More info on the subject can be found in [11]

- Microsoft Enhanced Cryptographic Provider - it is similar to the previous one, but provides additional algorithms thus increasing security, however is restricted by the US government
- Microsoft DSS Cryptographic Provider - supports signature verification and electronic data signing by using SHA and DSA algorithms. It is not restricted
- Microsoft Base DSS and Diffie-Hellman Cryptographic Provider - supports similar function as the previous one, provides additionally Diffie-Hellman key exchange. It is restricted
- Schannel Cryptographic Providers - support data integrity, session key exchange, and authentication while using SSL and TLS protocols. These providers are not restricted by any international organizations

It can be noticed, that providers, which support better cryptographic functions, are usually subjects to United States government cryptography export restrictions. A question arises, what kind of functionality is provided by both kind of CSPs. It can be shown on example of the Microsoft Base CSP and Microsoft Enhanced CSP. A table below shows different algorithms used by those providers, as well as supported key lengths:

Algorithm	**Base CSP**	**Enhanced CSP**
	key length in bits	
RSA public key signature algorithm	512	1024
RSA public key exchange algorithm	512	1024
RC2 block encryption algorithm	40	128
RC4 stream encryption algorithm	40	128
DES	N/A	56
Triple DES (2-key)	N/A	112
Triple DES (3-key)	N/A	168

It can be easily noticed, that the Microsoft Enhanced Cryptographic Service Provider provides better security and variety of different cryptographic algorithms in comparison to the Base Cryptographic Service Provider.

To conclude this chapter, it may be said, that the Microsoft Windows operating system has been well equipped with tools, which support many cryptographic functions which fulfill different roles, such as encryption and decryption of data, digital document signing, hashing, key exchange, authenticity verification and many more, and can really satisfy user's needs.

4.2 Cryptographic services under Linux OS

Similarly to the Windows operating system, many cryptographic libraries have been implemented to work under Linux operating system. Most of them support such basic cryptographic algorithms, as for example RSA, DSA, MD5, SHA-1, AES, and many more. The user may choose among several of the available cryptographic libraries, and his choice may depend on the license (if the use of a library is restricted in any matter), and the language, which the library has been written in.
The following cryptographic libraries may be used under Linux operating system:

- Crypto++ - written in C++, includes ECC functions

- OpenSSL - written in C, supports SSL and TLS protocols - similarly to Schannel Cryptographic Providers distributed under Windows
- CryptLib - written in C, includes hardware supports and self-tests
- Cryptix - written in Java, supports SSL and TLS protocols
- MIRACL - written in C/C++, supports many cryptographic functions including ECC and Lucas functions
- Flexiprovider - written in Java, supports ECC and advanced hashing functions
- OpenCL - written in C++, supports most of major cryptographic functions

Of course, those are only most commonly used examples of cryptographic libraries, which may be used under Linux. This operating system is very flexible, when it comes to developing and adapting various types of software, including cryptographic libraries.

4.3 Cryptographic services under Mac OS

Similarly to the previously described operating systems, certain cryptographic libraries can be used under Mac operating system. Of course, the choice is not as big as among the two of the most popular operating systems, but the user may still find expected support.
Here are some of the libraries, which are believed to be most often used under Mac operating system:

- PowerCrypt - supports following cryptographic functions: DES, RSA, DSA, PKCS, MD2, MD5; e-mail protocols: PEM, S/MIME; creates and stores keys and test certificates
- Crypt - library used for encryption and decryption of files. Supports Blowfish algorithm
- EnScript - similarly to the Crypt library, it provides good encryption methods, supports Blowfish algorithm

It can be noticed, that the Mac OS has also been equipped with cryptographic libraries, which support all necessary cryptographic functions, including main encryption algorithms, hashing, e-mail protocols, and file encryption and decryption.

5. The Electronic Signature Bill *(Robert Jastrzębski)*

5.1 Introduction

Regulations describing all aspects of the electronic signature have been included in a bill from 18 September 2001, released on 15 November 2001 at "Dziennik Ustaw Rzeczypospolitej Polskiej Nr 130". The bill's number is 1450. It consists of several chapters, each of which consists of a number of articles. Only most important, from the point of view of the project, will be presented and commented.

5.2 Abstract

Chapter I - General regulations

Art. 1. The bill specifies conditions of using electronic signature, its law consequences, and provision of certificate services and rules of supervision of entities providing these services.

Art. 3. Expressions used in the bill mean:
1) electronic signature - data in electronic form, along with other data, to which it has been connected, or on which it is logically dependent, which is being used to identify a person, who puts an electronic signature,

4) data used to put an electronic signature - data, which is unique and has been assigned to a person, who uses it to put an electronic signature,

5) data used to verify an electronic signature - data, which is unique and has been assigned to a person, who uses it to identify the person who puts an electronic signature,

10) certificate - an electronic certificate, which assigns data used to verify an electronic signature to a person who puts an electronic signature, and which allow to identify this person.

Chapter II - Law consequences of electronic signature

Art. 5.2. Data in electronic form being signed with an electronic signature, which can be verified by a valid certificate, is equivalent to a document being hand-signed.

Art. 5.3. An electronic signature, which can be verified by a valid certificate, assures integrity of data signed with this electronic signature in such way, that every modification of this data is recognized.

Art. 6.1. An electronic signature, which can be verified by a valid certificate, proves, that it has been put by the person specified as a person authorized to put this electronic signature.

Chapter IV - Providing certificate services

Art. 15. The receiver of certificate services is obliged to keep data used to put an electronic signature in a way assuring its protection from unauthorized use.

Art. 20.1. A certificate contains at least the following data:

1) certificate number

3) name of the provider of certificate service

4) name and surname or an id of a person putting an electronic signature

5) data used to verify an electronic signature

6) certificate validity and expiry date

7) electronic authentication of the provider

Chapter VIII - Penalty regulations

Art. 47. A person, who puts an electronic signature, using data used to put an electronic signature, which has been assigned to a different person, is a subject to a fine, imprisonment up to 3 years, or both.

Art. 52.1. A person, who is obliged to keep certificate services in secrecy, discloses or makes use of reserved information, is a subject to a fine up to 1000000 zlotys, imprisonment up to 3 years, or both.

Chapter IX - Final regulations

Art. 58.1. Banks and public authority institutions, will adapt their services to provide electronic certificate services up to 31 December 2002.

5.3 Conclusions

Chapter I of the bill specifies and describes certain keywords, which will be later used in the bill. Translating them to an understandable language, it may be said, that data used to put an electronic signature is simply a private key, and data used to verify an electronic signature is a public key.
Chapter II is very important from the point of data encryption and authentication. Article 5 states that data signed with an electronic signature is equivalent to a hand-signed document and that after verification every modification of data is recognized. Article 6 states that an electronic signature, which can be verified by a valid certificate, proves, that it has been put by the person authorized to put such signature. Probably, this is the most important article of them all.
Chapter IV describes fields of a certificate, to which surely belong: its number, name of its provider, id of the user, his public key and certificate's validity period.
Chapter VIII regulates penalties, which will be imposed on a person, who pretends to be another certificate owner - in other words fabricates an electronic signature. They are extremely high, namely up to 3 year of imprisonment and fine up to 1 million zlotys.
Chapter IX states, that all banks and public authority institutions, should adapt their services to provide electronic certificate services up to 31 December 2002.

IV. Project Description

1. Our Choices of Implementation *(Michał Poręba)*

As a practical part of the thesis, after consulting the supervisor and taking into account the results of the analysis presented above, we have decided to implement the sample core of the system in the following form:

1.1 General Method

A client-server method described as 'online' in the chapters above. An XHTML document format for data presentation, an XML-embedded XHTML for document storage, user data submitted using standard HTTP mechanisms.

1.2 Client

Client system is basing on HTTP protocol, using standard web browser for receiving the data, processing it client-side and submitting it to the server. An ActiveX plugin for MS Internet Explorer™ (most widely-used web browser in Poland) is written, allowing to sign the document before submitting it to the server. The plugin may be downloaded from the server if not yet installed on the client machine.

1.3 Server

The server system is based on Apache II web server with PHP module installed, PHP server-side scripting[1], software written in C language and MySQL database engine. This set of technologies allows for a very large scale of portability – after a small number of adjustments (regarding mostly file system structure) the server can be moved from Windows™ to Linux or UNIX ™ operating system or the other way around.
The test server platform is AMD Duron (800MHz) PC with Windows™ 2000 Professional™ operating system.

1.4 Scope of the Project

Since the system is not a specific implementation for a given company or agency, rather a sample one designed to be customized to the specific needs of the user, the following areas have been considered out-of-scope for the implementation part:

- Printing and OCR of the non-electronic document. Such a document is not an e-form in a strict sense.
- Secure link between client and server. Standard SSL would be employed to encrypt the communication between client web browser and the web server.
- Digital certificate for the office's web page, sending signed confirmation to the user. These parts would also include using third-party software and would be useful only in the environment of the specific implementation.

Additionally, the process of checking the digital signature is reduced to consulting local 'repository' stored in a text file instead of connecting to the real CA. On the client side, the keys are entered by hand into the form displayed by the plugin.

[1] PHP specification: [14]

2. System Architecture *(Michał Poręba)*

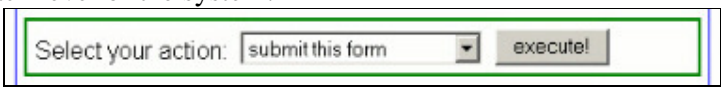
2.1 General Appearance

The system is a rather typical web application, working in a client-server environment. This means that the user interacting with the system will perceive it as a series of web pages to navigate through. This interaction will work the following way:

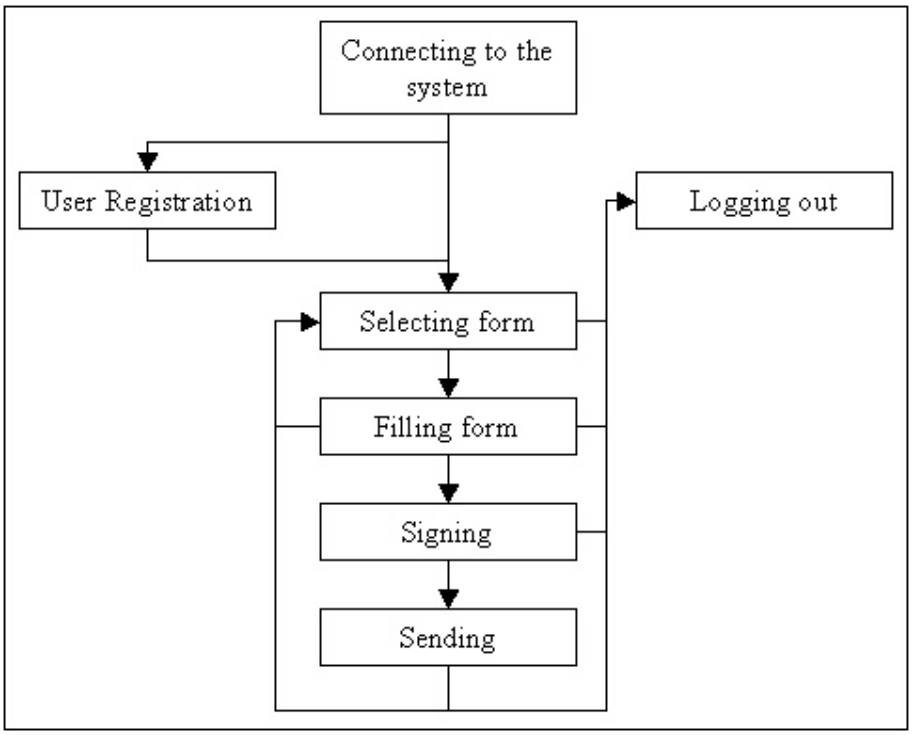
- Before being allowed to do anything at all, a new user must register in the system database, entering some basic data and being assigned a unique identity and password.
- After registering, the following phases can be accessed consecutively:

- Logging in
- Selecting an e-form to be filled
- Filling the form
- Submitting the form for data validation
- Receiving the document to be signed
- Signing the document
- Submitting the document
- Receiving confirmation about acceptance of the document

User can navigate through the pages by using a set of menus, presenting possible choices on each level of the system:

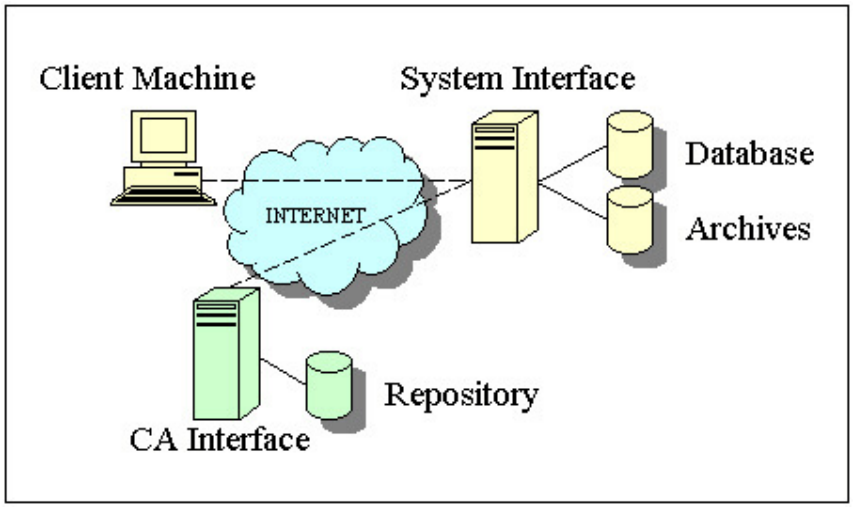


General navigation structure can be presented as follows:



2.2 Modular Composition

All the functions mentioned above are realized by a set of semi-independent modules written in different programming languages, divided in deployment between the client and server systems and communicating with each other.



In general, each of the modules is responsible for managing a single activity during the system operation. The system consists of the following modules:

Module	Responsible for	Side	Technique	Author
Registration and Login	managing users and authorizing their access to the system	Server	PHP, SQL, XHTML, JavaScript	Michał Poręba
Document Creation	letting the user to select and fill requested form, validating form data, generating the document to be signed	Server	PHP, SQL, XHTML, JavaScript	Michał Poręba
Document Signing	signing the document	Client	ActiveX, C++, Visual Basic	Robert Jastrzębski
Document Submitting	submitting the document to the server	Client	XHTML	Michał Poręba
Document Receiving	receiving the document submitted by user and sending it to verification	Server	PHP	Michał Poręba
Document Decoding and Verification	decoding the document and assessing signature validity	Server	C++	Robert Jastrzębski
Data Archiving	saving the signed document and updating data in database	Server	PHP, SQL	Michał Poręba

2.3 Server Environment

For the demonstration purposes of this project, the server had been set up and configured in the following way:

Web Server

As mentioned before, the Apache HTTP Server is used as a web server. This open-source project managed by the Apache Software Foundation is currently number one web server in the Internet – in July 2003, an estimated 63% of the web sites were being served using Apache in any of its version and under various operating systems. In the project, Apache version 2.0.45 is employed.[1]
On the test server, the Apache is installed in the directory *C:/Program Files/Apache Group/Apache2*.

PHP

PHP ('PHP: Hypertext Preprocessor) in version 4.3.1 is installed as a server module. PHP is a server-side scripting language destined for Web development that can be embedded in HTML and gains more and more popularity these days. It is developed as an open-source project; more info regarding PHP can be obtained from http://www.php.net.
On the test server, PHP is installed in the directory *C:/Program Files/php*.

MySQL

Another well-known open-source project, MySQL (http://www.mysql.com) is a relational database that is widely used throughout the world. While it lacks some power with respect to the large commercial database engines (e.g. it doesn't support nested SELECT queries), it is useful enough to be employed in many commercial and non-commercial enterprises. If the security, functionality or reliability level offered by this database would be too low for a desired purpose, the database engine can be switched to anything from PostgreSQL to any of the commercial ones.
On the test server, MySQL is installed in the directory *C:/mysql*.

2.4 File Structure

All of the project-inherent files are installed in one directory – on the test server it is *C:/users/mpr*. This directory is also pointed to by the Apache configuration *DocumentRoot* directive, making it a server document root – a default starting point for any user connecting to the server via the HTTP protocol will be an *index.php* file stored in this directory.

Basic files

All main script files are stored in this directory. These are:

File name	Module/function
locale.cfg	a system configuration file
style.css	CSS style sheet for the system's web pages
form.php	form selection, filling and submitting
form-out.php	validating form data, generating ready-to-be-signed document, initiating signing process, uploading signed document
index.php	user login

[1] For more information on the Apache HTTP Server, see http://www.apache.org and http://http.apache.org.

logout.php	safe logging out script
register.php	user registration
upload.php	document receiving, initiating document verification, data archiving

The functionality of most of the files is described in the sections devoted to the appropriate modules, further in this document.

The *locale.cfg* file

This file contains some path definitions that make it easy to modify the project file structure and server configuration.

```
…
//###### http path from outside to the project root ######
define("HOSTPATH", "194.29.168.139");
…
//###### local disk path to root of the project ######
define("DISKPATH", "c:/users/mpr");
…
//###### repository file ######
define("REPOSITORY", DISKPATH.'/repository/rep.txt');
…
```

Few of the useful definitions presented in the except above are:

- The HOSTPATH constant defines the absolute URL to the project root directory on the server. If any of these is changed (for example host's IP or name is changed), this value must be appropriately adjusted in order for the system to work properly.
- The DISKPATH constant defines the absolute local path to the project root directory.
- The REPOSITORY constant defines the filename of the local 'key repository' file – see the section on server-side digital signature verification further in this document for more information.

Directories

A number of subdirectories are used to store the system's subprograms, downloadable modules, source data etc.

Directory name	Contents/purpose
decode	an .exe file responsible for document decoding and signature verification
forms	source data for available forms
functions	PHP definitions of functions used
plugin	a .cab file containing browser plugin
puzzle	non-dynamic forms and page elements that are to be included into the script-generated web pages
repository	a file simulating CA repository
sessions	temporarily stored users session data
temp	temporarily stored user-filled forms
uploads	temporarily stored user-uploaded signed documents
users	submitted forms archive

3. User Registration and Login *(Michał Poręba)*

3.1 User Interaction

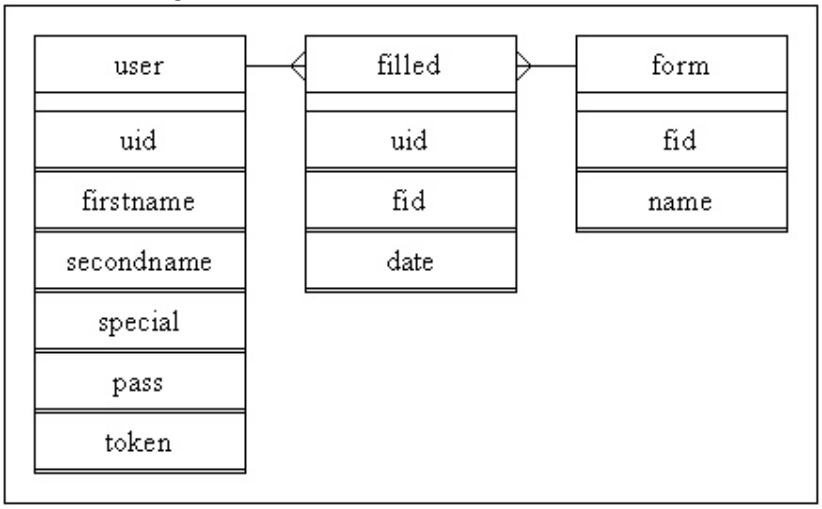
Every user in the system is identified by a unique ID number (dubbed 'UID' for the purpose of this implementation). When a user is entering the system (i.e. opening the system's main web page), he is asked about his UID and password. He is also offered an option to register as a new user.
When registering, the user has to supply his PESEL number, which from now on will be treated as his unique ID. He is also asked to define a password that will be used for logging in. If all the information is proper and valid, the user is added to the database and may log into the form filling system.

3.2 File Structure

This entire module is contained in two basic script files. *Index.php* is the front-end for the system. It is called every time a new user connects and allows selection between logging in and registering. It also calls authorization routines for user logging in and – if the authorization had been successful – redirects user to the next module (managed by *form.php*). *Register.php* file is responsible for registering new users, after the registration process is done, it redirects back to the front-end page.

3.3 Holding User Information

In order to keep track of the users and the e-forms that have been submitted, a simple relational database is used. This database holds basic user information, authorization data and details about the forms that are available in the system along with information about which of them has been already filled and submitted by a specific individual. An E-R Diagram for this database looks as follows:



3.4 Maintaining Application Context over HTTP Protocol

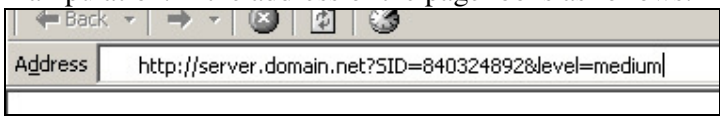
The HTTP protocol was designed in such a way that for every request (e.g. asking for returning a specific web page) a new connection with the server is initiated. When the download is complete, this connection is ended[2]. Due to this, the HTTP protocol –

[2] In HTTP version 1.1, this was changed to allow downloading a number of files in a single TCP connection, but then the connection is broken anyway and a new one must be established for the next request.

unlike, for example, telnet or ftp – does not offer any mechanisms to save the state of the application. This stateless nature of the protocol means that every data that requires persistence (e.g. the specific user identity and the current state of his session with the application) must be handled and recorded by the application itself. A number of ways of doing it have been devised, most common of them being the propagation of the session id in URL and the usage of cookies.

URL session propagation

In this method, a unique variable called 'session ID' is generated for every user logging into the system/application. This variable is then passed in URL every time the user clicks on some link in the pages he navigates, so every new page is being passed the session ID and can identify the user. In a similar way, other data requiring persistence may be passed via the URL. This method is relatively simple, but prone to manipulation. If the address of the page looks as follows:



any user can substitute any value for the session ID (*840324892* in the presented example) and present himself to the application as a completely different person. Of course, in most cases, there will be no active user with such a SID, and therefore nothing disastrous will happen, but creative usage of such a method can lead to taking control of another person's account on the server[3].

In some cases, this vulnerability could be used not only to fake 'client-submitted' variable values, but also to overwrite some global range variables used in server-side scripts for internal purposes only – for example, standard configuration of PHP up to version 4.2 was not safe against such an attack.

In order to prevent user tampering with the session and other variable status, an alternative method of passing data can be used.

Cookies

An HTTP standard expansion first introduced by Netscape, cookie is a tiny text file that is stored on a client machine. A web application can store some data (e.g. the session ID, but also user identity, visit history, personal settings etc.) in the cookie and retrieve it in the next HTTP connections. This method is more difficult to hack directly – of course, the cookies are stored on the disk and can be changed or intercepted by any user having access to these files, but this requires a bit more work. Cookies can also be encrypted, which makes them fairly safe from manipulation, if not interception.

Implementation

In the designed system, the method employed for maintaining session and application context is using cookies. When the user connects to the system, he is being sent a cookie containing his session ID. After the successful logging in (UID and password supplied by the user were verified), the user is issued a 32-byte long random token that is stored in the cookie and in the appropriate field of the user record in the database. This token, along with user name also remembered in the cookie is used in the next connections of the same session in order to avoid storing the password in cookie and/or submitting it more times than it is absolutely required – which could

[3] In fact, such an attack is known to have succeeded at least once against each one of the three largest Polish free email account providers.

lead to the password compromising. At the beginning of every connection following the first one, user name and token are read from the cookie and checked against the database. If the token is valid for the given user, his identity is confirmed and, simultaneously, he is granted an access to the requested page.
After user logging out, the session cookie is destroyed, and the token is deleted from the database.

3.5 Safety Concerns

Since the system is using a digital signature in order to verify the user's identity when submitting the form, this preliminary logging in is just a 'first line of defense', introduced rather for organizational (context maintaining, as described above) than security reasons. Nonetheless, necessary precautions must be made in order to hack-proof the system and prevent at least the most popular methods of attack.

SQL Injection

SQL Injection is one of the easiest methods to bypass authorization procedures and gain access to the protected assets. It can be used against systems that hold the user names and passwords in an SQL-using database and check the password by sending an appropriate query to the database.
The simple password checking script (written in PHP) may look as follows:

```
$result=mysql_query("SELECT * FROM user WHERE username='$uname' AND
password='$pass'");
$num_results=mysql_num_rows($result);
if($num_results!=1) error('User not authorized!');
```

Here, the first line is responsible for constructing the SQL query that will be submitted to the database engine. *$uid* and *$pass* are variables containing data entered by the user. In normal situations, the user would simply enter his UID and password.
If the result of the query would consist of one and only one row (i.e. one and only one 'user-password' matching pair have been found), the authorization would be considered successful.
However, such a mechanism can be easily fooled. The user can for example enter the following data:

```
Username: Cal'vin
Password: qwerty
```

The resulting SQL query built from such a data and sent by the script to the database will look as follows:

```
SELECT * FROM user WHERE username='Cal'vin' AND password='qwerty';
```

This will of course generate an error and nothing disastrous will happen, but other commands may include for example:

```
Username: blah'; drop table user--
Password:
```

In this case, the query will look as follows:

```
SELECT * FROM user WHERE username='blah'; drop table user-- AND
password='';
```

Since the 'double hyphen' is used for marking a single line comment in Transact-SQL, the result of execution of such a query will be dropping an entire 'users' table (of course only if the appropriate privileges are granted to the PHP script), which leads to a crash of the entire system. By using this mechanism, one can not only crash the system, but also log in as an arbitrary user, change any user's password etc.

Faking the HTTP request

Another, more advanced technique of intrusion is faking the data sent in HTTP request – be it cookie content, form data or any other variable content. User can not only tamper with the cookies that are saved on his disk or manipulate the URL – he can generate his own HTTP requests (by using telnet on port 80 or, more practically, generating own web pages/scripts) containing his own data (including fake cookie content), pretending to be the data expected by the server-side script from a web page generated by this script. Because of this, absolutely no information coming from the user can be treated as absolutely safe – all the parameters and values have to be treated as 'user-originating' and checked in a similar way as the data 'more obviously' entered by the user.

Precautions taken

In order to protect the database against such types of attacks, the following precautions have been taken:

- 'PHP user-agent' that is connecting to the database had been granted minimal privileges that are necessary.
- Input fields are set a limited length that won't allow for entering too long queries (but this can be bypassed by faking the HTTP request, see next subchapter).
- A set of special functions is written for analyzing every data entered by user and checking whether they are of proper type and don't contain illegal and potentially dangerous character sequences.
- Critical data arriving from user is checked in a similar way, especially the data that is used to access the file system or database.

4. Document Creation (w/data verification) *(Michał Poręba)*

This module is based on the *form.php* file and its main goal is to create the document that will be ready to be signed by the user. This process of creating a ready-to-be-signed document from a blank e-form is composed of a set of consecutive stages. First, the proper e-form is selected and downloaded by user. Then, the form is filled with data. Then it is submitted for data verification. If some of the data is found incorrect, the form is returned to the user to be corrected. Otherwise, an XHTML document is generated basing on the form data and containing all the information previously entered. This document is saved on a server in a temporary folder for the purpose of later verification, and a copy is made available for the user to be downloaded.
The user can either download the file using web browser, or run the signing plugin that will download and electronically sign the file by itself (see the chapter on Document Signing).

4.1 Form Selection

After having successfully logged into the system, the user is presented with a list of all the e-forms that are available. This information is generated by a PHP script that contacts the local database, checks which forms have been already filled by the active user, and generates the form list appropriately. Forms already submitted have their names displayed, while forms not yet completed can be selected for filling.

4.2 Form Filling

An e-form is rendered using standard XHTML <*form*> tag and its elements, as inherited from HTML syntax. Sample code of such a form can look as follows:

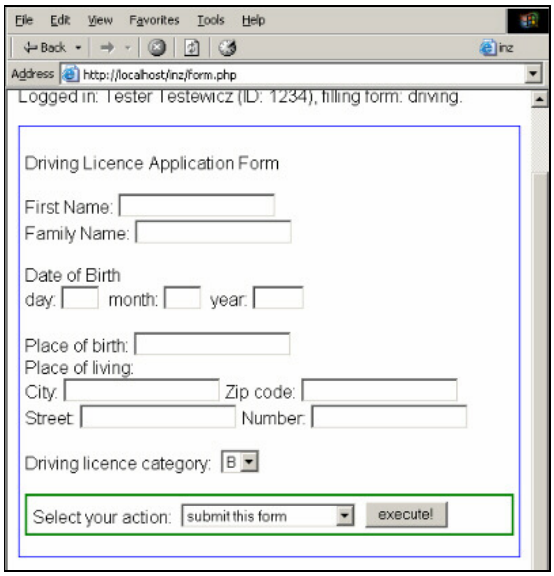
```
Driving License Application Form<br />
<br />
First Name: <input type="text" name="firstname" maxlength="30" size="20" /><br />
Family Name: <input type="text" name="familynamename" maxlength="30" size="20"
/><br />
<br />
Date of Birth<br />
day: <input type="text" name="birthday" maxlength="2" size="2" /> 
month: <input type="text" name="birthmont" maxlength="2" size="2" /> 
year: <input type="text" name="birthyear" maxlength="4" size="4" /><br />
<br />
Place of birth: <input type="text" name="birthplace" maxlength="20" size="20"
/><br />
Place of living: <br />
City: <input type="text" name="city" maxlength="20" size="20" />
Zip code: <input type="text" name="zipcode" maxlength="20" size="20" /><br />
Street: <input type="text" name="street" maxlength="20" size="20" />
Number: <input type="text" name="number" maxlength="20" size="20" /><br />
<br />
Driving license category: 
<select name="category" class="field-meni">
<option value="A">A</option><br />
<option value="B" selected="selected">B</option><br />
<option value="C">C</option><br />
<option value="D">D</option><br />
<option value="E">E</option><br />
</select><br />
```

This e-form's sources are stored in separate files in the *forms* directory. For every form, two such source files exist. First of them (form_id-*src.inc*) contains an XHTML

description of the form as presented above, the second one (form_id-*val.inc*) contains specific data verification rules that are to be enforced on the data that was entered by the user. When the user selects a specific form, an appropriate form identifier is submitted to the server and is used to generate the path to the source files. Since this identifier is passed from the user, its value is checked every time it is used in any file-based operation – if not for this, any file from the server could be potentially accessed by this method.
After the identifier is accepted as valid, the form code is extracted from the file and inserted into the generic page code by a standard PHP *require()* function to produce the web page code that is relayed to the client's web browser. Following the example above, this code would look as follows (most parts of the XHTML code responsible for defining the appearance of the of the page have been omitted for clarity):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "DTD/xhtml1-
transitional.dtd">
<html>
<head>
…
</head>
<body>
…
<hr noshade="noshade" />Logged in: Tester Testewicz (ID: 1234), filling form:
driving.<br /><br /><div class="form"><form action="form-out.php"
method="post">Driving License Application Form<br />
<br />
First Name: <input type="text" name="firstname" maxlength="30" size="20" /><br />
…
Driving license category: 
<select name="category" class="field-meni">
<option value="A">A</option><br />
<option value="B" selected="selected">B</option><br />
<option value="C">C</option><br />
<option value="D">D</option><br />
<option value="E">E</option><br />
</select><br /><br />
<div class="action">
        Select your action: 
        <select name="formaction">
                <option selected="selected" value="submit">submit this form</option>
                <option value="save">save form to disk</option>
                <option value="load">load saved form</option>
                <option value="clear">clear all data</option>
                <option value="fselect">return to form selection</option>
                <option value="logout">log out</option>
        </select> 
        <input type="submit" value="execute!" class="action-submit" /><br />
</div>
</form>
</div>
<br /><hr noshade="noshade" />
…
</body>
</html>
```

After this data is interpreted by the user-agent, the following e-form may be presented to the user:



This form can now be filled in a standard and well-known way, encountered in many web applications, simple Internet quizzes etc.

4.3 Form Submitting and Verification

Submission

After filling the form, the user will probably decide to submit it. As in other places when user-entered data is submitted to the system, it is done using the POST method of HTTP protocol. The target of the data sent from this form is the *form-out.php* script, which is responsible for analyzing the entered data and reacting to its contents in an appropriate way.

Verification

First, all the submitted data that is critical for the system work and security is checked against suspicious or illegal patterns. Then, an appropriate block of code for the verification of the data coming from this specific form is included from the appropriate source file, using the *require()* function.
If there are any errors found, the user is redirected back to *form.php* script, with appropriate information regarding missing or invalid fields of the form.

Preparing to the Signing

If all the previous steps are concluded successfully, the script generates the output document and stories it in a temporary location on the server (this location is unique for each user session). The same data but in a slightly different form is written to the web page that is presented to the user in order to allow him to review and confirm its semantic validity for the last time.
A link to the stored document is also presented in order to allow the user to see the document in exactly the same form as it is going to be signed in. Additionally, this 'stored' form of the document will be subject to printing if the user decides to do so.
An ActiveX plug-in is also included in the resulting page. If the plugin is not yet installed, appropriate information will be displayed and, depending on the browser's security settings, the plugin may be downloaded either automatically (as most of the

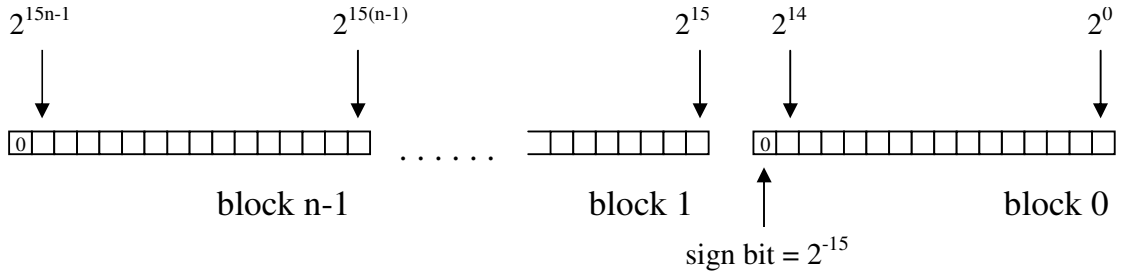
XXX pages dialers do), or after user confirmation. The plugin will also be updated if a new version is present on the server.
User has to click on the VBScript-generated button to invoke the signing procedure.

5. Document Signing *(Robert Jastrzębski)*

5.1 C++ implementation[4]

The concept

Since typical hashing functions, like MD4 and MD5, produce o 128-bit output and the message, which the RSA algorithm works on, is a hash of a document, it is necessary to operate on very large numbers (at least 128 bit long). Typical integer types supported by C++ programming language cannot satisfy the programmer's needs, because the *integer* type is four bytes long, and on some machines the *long integer* type covers eight bytes. The conclusion is that all of the integer types supported by C++ have insufficient length and it is necessary to define a completely new type.
The idea is to combine long numbers from blocks of short integers. Since a variable of type *short* has 16 bits of length (only 15 are used in this approach, 1 bit is reserved for sign and is not used in the calculations) it is enough to use 9 blocks to get a 128 bit long number.



This figure illustrates a sequence of *n* blocks. Having 9 blocks means that *n* equals 9 in the $(n-1)^{th}$ block and the total number of bits covered by all of them is equal to 135. Although this approach looks very convenient, it may not be that easy to implement, namely the concept of operator overloading must be introduced.

Operator overloading

Operations on such structures of blocks require introducing of completely new operators. For example, if we want to add two numbers it is insufficient just to add values of all corresponding blocks and store the result, because an overflow can occur. To prevent this from happening we must overload the + operator in such way, that the carry flag would be set whenever necessary and each block operation should take into account this flag from the previous addition. In a similar manner all necessary operators such as multiplication, subtraction, division, modulo, equality check, etc. should be implemented.

Code explanation

The MyInt structure
Before we can implement the structure, definition of constant *LENGTH* is required:

[4] See [13]

```
#define LENGTH n
```

where *n* is the number of blocks necessary for performing correctly all operations. Of course, some safety precautions have to be taken into consideration, since for example while multiplying two 128-bit numbers we may expect the result to be at least twice as long. So the value of *n* should start from about 24 or even better 32. The *MyInt* structure looks as follows:

```
struct MyInt {
  short block[LENGTH];
  short last;
};
```

There are two fields of the structure. First one is a table of elements of type *short*. The previously defined LENGTH constant determines its size. The second field is a variable called *last* of *short* type. It is responsible for storing the number of last occupied block. Its role is important, since speed of the calculations is crucial for the RSA algorithm. For example, having to add two numbers, each 2 blocks long, would take unnecessary *LENGTH* operations to complete. Why the type of block has been chosen as *short*, not as *int*, will be explained in the next chapter.

operator+

The result of addition of two variables of any given type should give a third variable of the same type. However, operator overloading allows us to interfere with this process, to implement a correct algorithm for addition of two variables of the *MyInt* type. The *operator+* is defined in such way:

```
MyInt operator+(MyInt, MyInt);
```

After implementation of this operator, the following call should be possible:

```
MyInt a,b,c;
c=a+b;
```

The body of the operator is the following:

```
MyInt operator+(MyInt a, MyInt b) {
MyInt c;
short i,carry = 0;
        c.last = __max(a.last, b.last);
        for(i=0; i<=c.last; i++) {
                if ((c.block[i] = a.block[i]+b.block[i]+carry)<0) {
                        c.block[i]+=MAXSHORT;
                        carry = 1;
                }
                        else       carry = 0;
        }
        if (carry!=0) {
                (c.last)++;
                c.block[c.last] = 1;
        }
return c;
}
```

The procedure takes variables *a* and *b* on input and returns variable *c*. At first, the maximum value of *last* parameters of the input variables is calculated, to determine how many blocks will be involved during the addition process. Then, starting from the first block, the additions of all corresponding blocks are being performed in a *for* loop. If it happens that the result of an operation is negative, it means that the value of the resulting block has to be adjusted and the *carry* flag is set to 1, otherwise it is set to 0. If the number is negative, it means that its first (sign) bit, which has the value -2^{15}, as the figure presented before shows, has been set to 1. To set it to 0, it is enough to add to the corresponding block the value 2^{15}. Thus, the value of the *MAXSHORT* constant has been set to $2^{15} = 32768$. After the loop has finished, final condition has to be checked, because it might happen, that after performing the last addition, the carry flag has been set to 1. If so, the value of the *last* parameter of variable *c* is set to 1.

Operator*

Similarly to *operator+*, *operator** is defined in a following way:

```
MyInt operator*(MyInt, MyInt);
```

The body of the procedure is more complicated, comparing it to the addition procedure, and looks as follows:

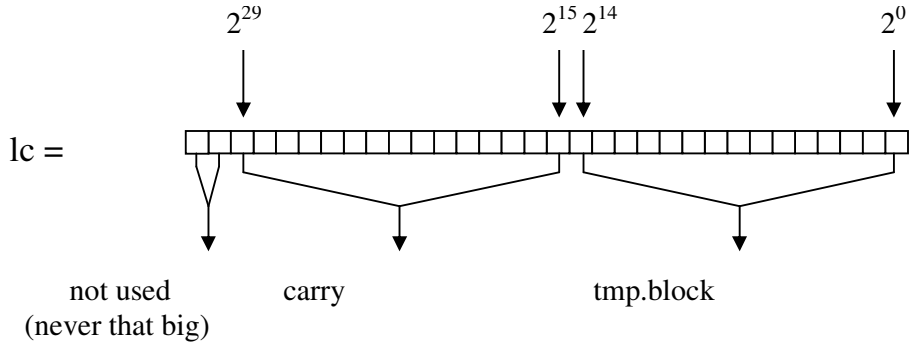
```
MyInt operator*(MyInt a, MyInt b) {
MyInt c,tmp;
short i,j,carry;
int lc;
        for(i=0; i<=a.last; i++) {
                erase(tmp);
                tmp.last = i+b.last;
                        carry = 0;
                for(j=0; j<=b.last; j++) {
                        lc = a.block[i]*b.block[j]+carry;
                        if ((tmp.block[i+j] = lc)<0) tmp.block[i+j]+=MAXSHORT;
                        if (lc<MAXSHORT) carry = 0;
                                else    carry = (lc>>=15);
                }
                if (carry!=0) {
                        (tmp.last)++;
                        tmp.block[tmp.last] = carry;
                }
                c = c+tmp;
        }
return c;
}
```

The whole code will not be explained, but some important points have to be mentioned. First of all, two loops are used instead of one. Moreover, two temporary variables are declared:

```
MyInt tmp;      /* and */          int lc;
```

The *tmp* variable is used for storing temporary result and is being erased before every execution of the second loop. The necessity of using an *integer* variable was the reason for choosing variable was the reason for choosing the *short* type for the array in the *MyInt* structure. Although the carry flag in the addition operation could be set at most to 1, when it comes to multiplication its value can be much higher. Since the

speed of the calculations is crucial, the multiplication is based not only on addition, since it would take enormous time to compute a product of two large numbers. Instead of this approach, each block of the first number is multiplied by every block of the second one (that is why two loops are introduced). The results are stored and further added to produce the result.



The temporary block stores the lower byte of the *integer* variable. If the value held by this variable is smaller than the *MAXSHORT* value, the carry flag is set to 0, otherwise it is calculated from the higher byte, namely by shifting the *integer* by 15 bits to the right. Although the multiplication algorithm looks quite complicated, it is really efficient.

Main crypto-function

The function has to compute the following equation:

$res = msg^{power} \% N$, where

- res - final message
- msg - initial message
- power - key E or D
- N - key N

In order to achieve this, a theorem must be introduced. Since the function will base on modulo operations, following statements will be very helpful for building the body of the function:

having a,b,c as integer numbers, it is true that since
$a \% c = (a \% c) \% c$,
the same is true for the product:
$(a*b) \% c = [(a \% c)*(b \% c)] \% c$

After defining those two statements, body of the function can be implemented:

```
MyInt crypt(MyInt msg, MyInt power) {
MyInt m,p,rem,one,tmp;
        one = 1; rem = 1;
        while(1) {
                erase(p);           p = 2;
                while(pow(msg, p)<N) p++;
```

```
                m = pow(msg, p);
                m = m % N;

                rem = rem*pow(msg, power % p);
                rem = rem % N;

                msg = m;
                power = power/p;

        if (msg==tmp || msg==one || power==tmp) return tmp;
                if (power==one) return (msg*rem) % N;
        }
}
```

The main *while* loop will continue until a quitting condition is met, namely, when the *power* of the equation will decrease to 1. Right side of the equation needs to be decomposed into a simpler form, since it is impossible to raise a big number to a high power directly. The trick is to decrease complexity of operations by the means of the above theorem. At first, the term msg^{power} is being decomposed to the following product:

$msg^{power} = msg^{p} * msg^{p} * ... * msg^{p} * rem,$ where

$p < power$ has been calculated in the second *while* loop and the reminder variable will be equal to the message raised to the remaining power - reminder of *power* and *p*:

$rem = msg^{power \% p}$

Now, the theorem will be applied to the new message and reminder:

since $m = msg^{p}$, then $(m*rem) \% N = [(m \% N)*(rem \% N)] \% N$

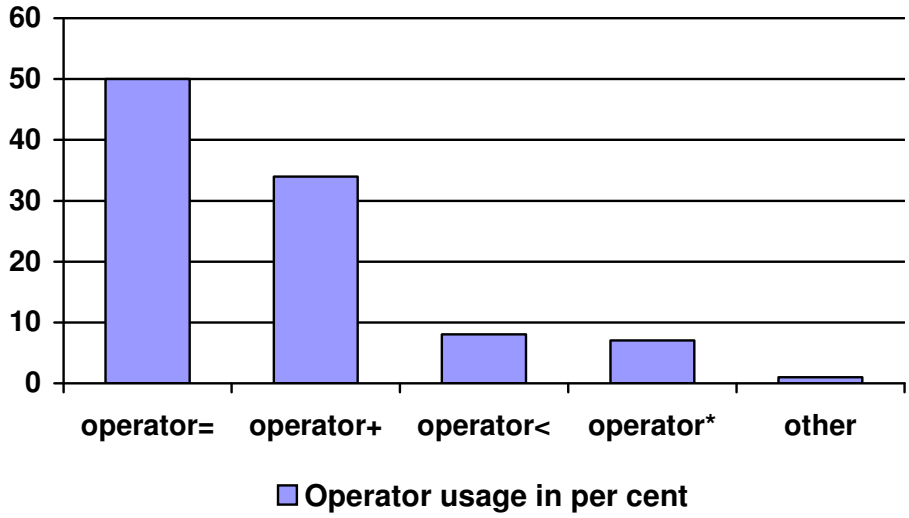
The final form to which msg^{power} had been decomposed looks as follows:

$msg^{power} = m * m * ... * m * rem,$ which in fact equals $msg^{power/p} * rem,$

having returned to the initial form of the equation, but in much simpler form, from the point of view of the size of components. The while loop will continue this process until one of the quitting conditions is satisfied. Either an error will occur, while supporting illegal keys (variable *tmp* of value 0 will be returned), or a correct result, namely *(msg*rem) % N* shall be returned as a result.

Operator usage

Since the speed of the algorithm is crucial for the project, it would be worth knowing, how many times each of the operators is being executed by the main crypto-function. Necessary operators, which have been implemented, are the following: addition (+), subtraction (-), multiplication (*), integer division (/), modulo (%), assignment (=), equality check (==), inequality check (!=), if smaller (<). A table below shows approximate operator usage during the calculation phase:



Most often used operator is the assignment operator. It is not surprising, since almost each operator function bases on some assignments. These operations take half of all needed to complete the coding. About one third of all operations are covered by addition. This operator is also very commonly used, especially in multiplication, and that is why it is so often used. Operator< is used quite often too, while it is needed by the modulo and division operators. Other operators cover barely 1% of them all. To conclude this, it might be worth marking, that the addition operator must be implemented with great caution, to assure that its speed would be optimized.

5.2 ActiveX

The purpose

ActiveX controls are simple OLE objects, which can be placed into other applications that support OLE technology. The abbreviation of OLE is Object Linking and Embedding. It is a distributed system and protocol, which allows to put (embed) an object from one application into another one. Such property can have many advantages, since sometimes it is impossible to achieve some goal, only by using tools supported by an application. This is when placing an OLE object (in this case an ActiveX control) might come in handy. Since two of the most popular web browsers, Internet Explorer and Netscape Navigator, support OLE technology, an ActiveX control may be embedded in a website and used without any problems.
The following question arises, if use of an ActiveX control is really necessary for the project. The answer is yes, because of the following reasons:

- since support of a web browser is necessary, it is convenient for the user to use only one application (in this case the browser)

- using a client application for signing the document require manual download and installation of this software by the user, and, as it was mentioned before, the user should interfere with the process as little as possible
- installation and possible upgrade of an ActiveX control and other necessary components is being done automatically, practically with no interference of the user
- since the signing process requires advanced computation technology, and cannot be realized inside the document, it is convenient to make the most of a dynamic link library, which can easily cooperate with the control
- finally, an ActiveX control can be designed with use of many applications, for example such as Visual Basic, which support many tools which make the creation process extremely simple

The above statements prove, that use of an ActiveX control in the project is really necessary, and may greatly improve its functionality.

Embedding

An ActiveX control is being embedded in an html document using the *OBJECT* tag. Since it must be recognized by the system's registry, its *CLASSID* must be specified, using the following convention:

```
<OBJECT CLASSID="CLSID:nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn"
```

Since the control's version has been designed with binary compatibility, it assures that its *CLSID* is a unique key. When the document is being loaded, the script must check if versions of the components require upgrading. Therefore, the location and version of the cabinet file must be specified, using the *CODEBASE* call:

```
CODEBASE="http://server.do/path/filename.cab#version=a,b,c,d">
```

Specified URL determines location of the cabinet file and the version determines the actual state of this file, if it is newer than the previously installed, the components will be updated. The letters from *a* to *d* stand for major, minor, custom and build part of the version. Typically, the first version of the cabinet file is 1,0,0,0. Two more parameters are required to properly display an ActiveX control on a web page:

```
<PARAM NAME="_ExtentX" VALUE="x">
<PARAM NAME="_ExtentY" VALUE="y">
```

Values *x* and *y* of parameters *_ExtentX* and *_ExtentY* determine width and height of the control being displayed. To end the object call, </OBJECT> tag must be applied.

Cabinet file architecture

Cabinet files are characteristic for Microsoft Windows systems, while *jar* files fulfill a similar function on Unix type systems. They are commonly used for packaging certain components (like controls, libraries, executables), in order to download and install them on the system. Usually, they contain an information file, which is responsible for the installation process. The cabinet file used in the project is called *signup.cab*. It contains following files:

- *signup.ocx* - the ActiveX control
- *hash_rsa.dll* - library containing hashing- and crypto-routines
- *signup.inf* - information file of the cabinet

Architecture of the *inf* file is very important, since it is responsible for correct installation of the control and dynamic link library. It contains sections, each of which is responsible for other functions. The body of *signup.inf* looks as follows:

```
[Add.Code]
signup.ocx=signup.ocx
hash_rsa.dll=hash_rsa.dll

[signup.ocx]
file=thiscab
FileVersion=1,0,0,0
RegisterServer=yes

[hash_rsa.dll]
file=thiscab
FileVersion=1,0,0,0
DestDir=10
```

The *Add.Code* section is responsible for determining which files are to be installed on the system. They will be installed in reverse order of appearance in this section. It is sometimes important, since some libraries may need to be installed before execution of other components. In this case, the library will be installed before the control.
The *signup.ocx* section is responsible for installation of the ActiveX control. First call (*file=thiscab*) means that the *ocx* file is attached within this cabinet file. It might happen that the *file* value would point to an URL address or other file location. The *FileVersion* determines the current version of the control file. The file will be installed on the system only if it does not exist or its older version is present on the system. This feature prevents unnecessary installation or download of components. The *RegisterServer* key marks that the control should be registered if it does not appear in the system registry file.
The *hash_rsa.dll* section is similar to the previous one, with the difference of parameter *DestDir=10*. This line makes the installer put the library in the default Windows directory of the system. It will be visible for the control file afterwards.

Signing cabinet files

Signing cabinet files is important, since the web browser's security level will determine if download of unsigned ActiveX components is allowed or not. Of course, the user may enable this property manually, but such operation is not recommended for two reasons. First of all, the user should interfere with any of the applications (in this case the web browser) as little as possible, the environment should be user friendly and should not cause the user any problems. Secondly, and more importantly, signed ActiveX components assure the authenticity of the control, and provide more secure interference of the components with the system. The user will be asked if he wishes to allow installation of the components, if he trusts that their source is secure.
The cabinet file used in the project will be given a test signature, which will be applied to it using the Microsoft Authenticode technology. This technology consists of programs for digitally signing code files as well as tools for checking if these files

have been successfully signed. The signing process consists of three major steps listed below:

1. creating a certificate
2. generating a spc file
3. signing the cabinet file

The first step is realized using the *MakeCert* application, which of course is a part of the Microsoft Authenticode technology:

```
MakeCert.exe -sv "key.pvk" -n "CN=Test Certificate" test.cer
```

The *sv* option specifies the output file, which is the private key used to sign the certificate. If it does not exist, it will be created and the future certificate owner will specify the key. The *n* option determines the owner of the certificate - this name will appear, while trying to download the signed file. In this case, it is simply *Test Certificate*. The final parameter stands for the filename of the certificate, which is to be created.

After creating the certificate file *test.cer*, an *spc* file must be made:

```
Cert2Spc.exe test.cer test.spc
```

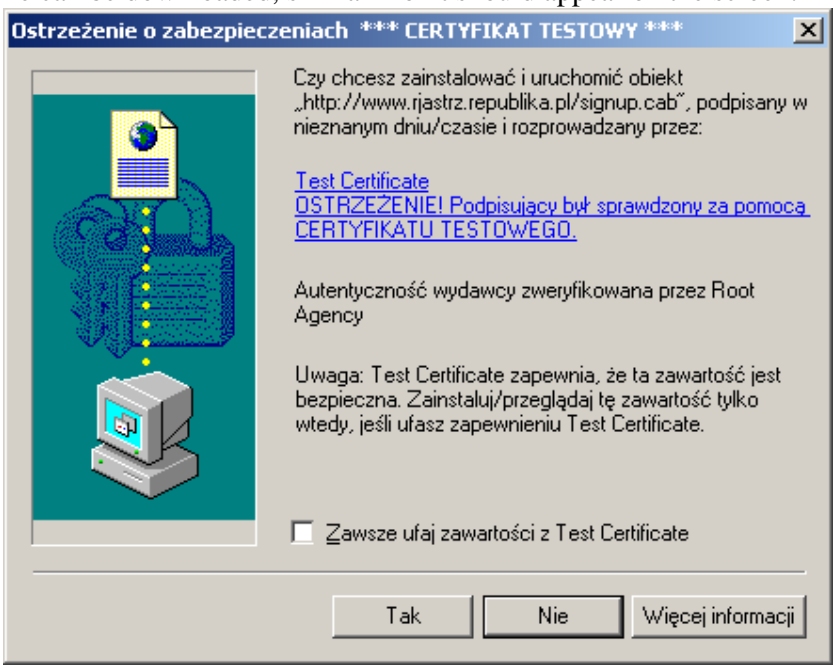
The *Cert2Spc* utility produces an *spc* file from a certificate. The *spc* file is so called PKCS #7 signed-data object, which contains information used to sign a file, usually the certificate.

Last, and most important phase, is the file signing step. The call used to perform this operation is as follows:

```
SignCode.exe signup.cab -spc test.spc -v key.pvk
```

The *SignCode* application allows us to sign the cabinet file. In this case, file *signup.cab* will be signed, using the created *test.spc* and *key.pvk* files. Of course, during the signing phase, the owner will be asked to input the private key value, which was set during the certificate creation step.

After applying this procedure, the cabinet file has obtained a digital certificate and is now ready to be used without any restrictions. Before a new version of the cabinet file can be downloaded, similar monit should appear on the screen:



This window shows that the cabinet file has been digitally signed with the use of the *Test Certificate* and the user can decide if he wishes to download and install it.

5.3 Visual Basic and Dynamic Link Library

The ActiveX control has been written in Visual Basic programming language. Visual Basic was chosen because of its certain properties, which made the programming easy. First of all, the control is being created directly at startup from the Project Wizard. Secondly, creation of forms and managing them is extremely easy, because all necessary visual tools have been implemented in the Visual Basic environment. Moreover, it is possible to use features of dynamic link libraries, which is indeed helpful in this case, since the hashing function and the RSA algorithm have been implemented in Visual C++ and covered by a dynamic link library.
The project consists of three major elements - the basic control button and two forms responsible for signing the document and choosing the output file. The document which is being signed is actually not the one on which the ActiveX control resides on. In fact, this document will be generated and held in some temporary location. A link to this location will be present somewhere in the main document:

```
<A id="form-out-result-link" href="http://server.do/temp/form_out.html">anything</A>
```

The URL on which the document is located has a unique *id*. It is important, since the control will have to 'extract' this URL from the document knowing only its *id*. It is being done in such way:

```
Dim path As String
path = Parent.Script.Document.getElementByID("form-out-result-link")
```

The location of the document is being assigned to the *path* variable, with the use of *getElementById* method and the *"form-out-result-link"* value.
The most important feature of this project is proper cooperation between the Visual Basic application and the dll. In fact, the crucial point is a correct passing of parameters between those two applications.
To be able to use any function from a dll under VB, it is necessary to include a *declare* statement into the code:

```
Private Declare Function enc Lib "hash_rsa" Alias "encrypt" (ByVal keyN As String, ByVal keyE As String, ByVal uid As String, ByVal foutput As String) As Long
```

Function *enc* is declared, its alias in the dll is function *encrypt*. The function takes four string variables as parameters and returns an integer value. The *ByVal* keyword assures that parameters are being passed by value, not by a reference. It is necessary to pass arguments of exactly the same type, for example *long* in Visual Basic and *int* in C++ are equivalent, otherwise problems may be expected. The function in the dll is built in such way:

```
int _stdcall encrypt(char* keyN, char* keyE, char* uid, char* foutput)
```

The *_stdcall* calling convention is necessary, because the compiler will now treat the function as a standard call of a Win32 API function, and the dynamic link library has been created in a Win32 environment. One more important thing must be included in the dll project. Namely, it is a definition file, including all functions, which can be accessible from outside the dll. In this case, it is a file *hash_rsa.def* having the following content:

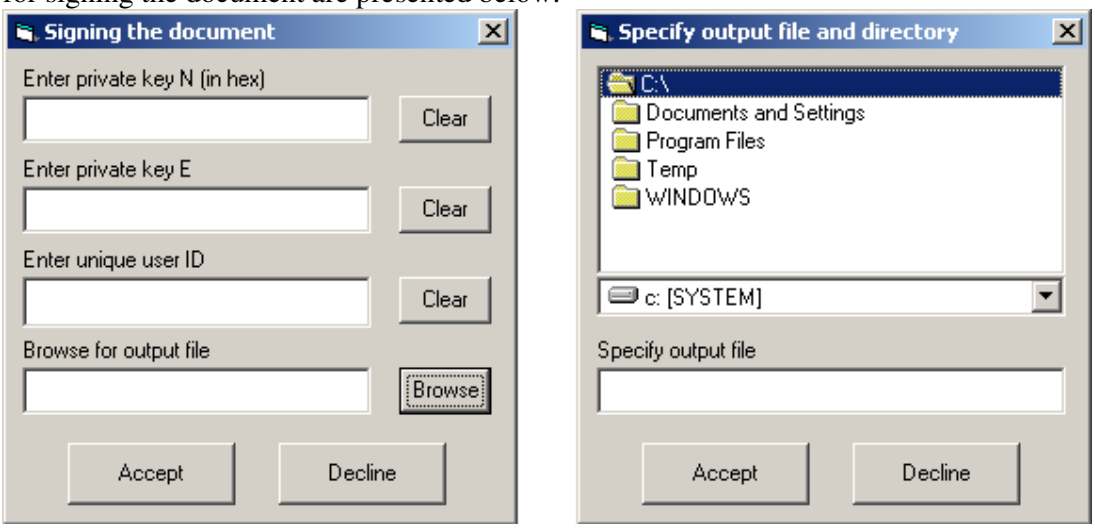
```
LIBRARY hash_rsa
EXPORTS
encrypt         @1
decrypt         @2
```

This code means that the library exports two functions named *encrypt* and *decrypt* respectively. The *@1* parameter determines that this function is the first function to be exported. The second function is used for testing purposes only, namely, to determine, if the hashing function and the RSA algorithm work correctly. The decryption will not be performed with help of a graphical user interface, since it must be accomplished manually from the server side.
Since the *encrypt* function returns a value, it is easy to display appropriate message boxes from the control in case of any problems, which may have occurred after some checking have been performed by this function. For example, the key values may be empty or not in a proper form (containing not allowed signs), data from input file cannot be read or the output file that was chosen already exists. Such feature is also very helpful during the debugging phase.
Part of the code of the *encrypt* function looks as follows:

```
if (keyN==NULL) return 1;                // checking if keys empty
if (keyE==NULL) return 2;
if (uid==NULL) return 10;
if (foutput==NULL) return 3;
if (assign_key(keyN,0)==1) return 4;     // assigning keys
if (assign_key(keyE,1)==1) return 5;
if (MDFile(foutput)!=0) return 6;        // hashing the document
```

```
text=hash_to_int();
if (N<text) return 7;                    // checking if keys proper
if (!(E<N)) return 8;
enc=crypt(text,E);
if (enc==tmp) return 9;
```

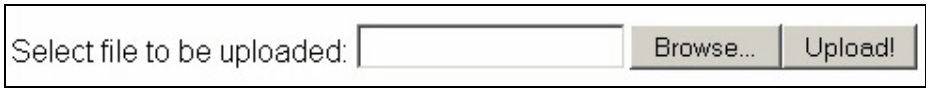
Appropriate values will be returned in case of any problems that might occur. For example: input strings are empty (1,2,3,10), keys are not in hexadecimal form (4,5), an error occurred during hashing (6), key N is smaller than the hash size (7), key N not greater than key E (8), invalid keys - function returned 0 (9). Having the return values, cooperation between the library and the dll is surely easier.
The forms fulfill basic functionality necessary for the key values input and correct choosing of the path and filenames of the files involved in the signing procedure. Thanks to the Visual Basic tools, the graphical view of the forms has been designed with ease, and it joins ease of use with expected functionality. The basic forms used for signing the document are presented below:



The form on the left is used to input public keys, user's id and browsing for the output file. The form on the right is obtained after pressing the *Browse* button and is used to specify the output file; the user may choose the directory by specifying a folder and a drive.

6. Document Submitting *(Michał Poręba)*

After the document is signed and saved to disk, the user has to select the proper file and submit it to the system. A mechanism to do this is included in the same page as the control button initiating the signing process.
File upload is realized using a special kind of a form field, designed specially for this purpose.



The code for this form looks as follows:

```
<form enctype="multipart/form-data" action="upload.php"
method="post">
<input type="hidden" name="MAX_FILE_SIZE" value="64000" />
```

```
Select file to be uploaded: <input name="fupload" type="file"
class="submit-window-button" /><input type="submit" value="Upload!"
class="submit-window-button" />
</form>
```

Clicking the 'browse' button invokes the operating system's standard file selection window. Clicking the 'submit' button sends the file (with specified maximum size) to the server via a defined HTTP submission method. In this case, POST method is used to transmit the file to the next module, which is handled by the *upload.php* script file.

7. Document Receiving *(Michał Poręba)*

A document submitted by HTTP POST method is to be saved and sent to the server-side software to verify its correctness.
Upon being uploaded, the file is saved in a unique directory. If no errors occur, the PHP script then invokes an executable program residing on the server that is written in C language. This program is to decrypt and validate the document, compare its contents with the file that was sent to be signed and return the results to the PHP system engine.
Main reason of doing it in C rather than PHP is the speed of execution. No matter which way of installation of PHP is chosen (as a web server module or using a CGI gateway), the speed of complicated numerical operations in parsed PHP is dramatically lower than the speed of any compiled language. Even when written in C and stored as a binary file, the program needs a significant amount of time to process the incoming file and check the signature and data validity (up to several seconds on an 800MHz Duron processor with Windows 2000 operating system and 512MB of RAM without any other software running), which leads to the conclusion that there would be no use in implementing these procedures directly in PHP.
The program is invoked using the *exec()* function, with parameters being filenames of the source files compared, temporary file and a local key repository (for the purpose of the thesis, Robert has decided to use a local file containing sets of keys and user data, instead of contacting the CA repository – see next chapter for details). Instead of commencing the user-entered data verification for the second time, the program simply compares the document that had been submitted recently with its copy saved on the server after the form filling completion. This saves both time and computing power of the system.

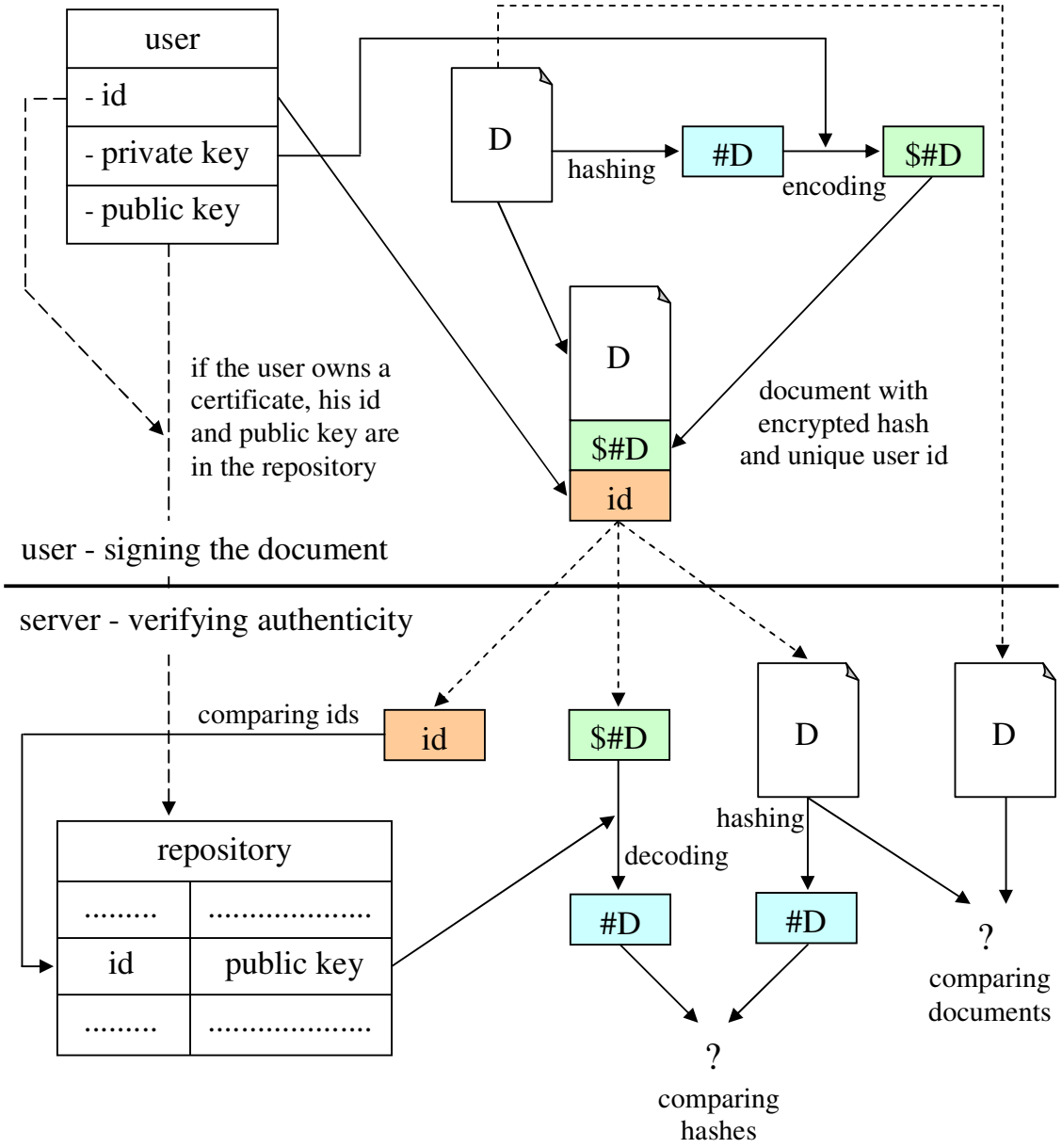
8. Document Decoding & Signature Verification *(Robert Jastrzębski)*

8.1 The overview

The main goal of the project is to verify, if the user, who logged in into the system, signed the document and sent it for verification, is really the one who he intends to be, and if the document sent is authentic for sure. It means, that the outcome of all phases included in the signing and verifying routines, should be the answer to the following questions:

1. Is the user the owner of a certificate?
2. Is the document sent for verification authentic?

A diagram below should help identify the problem and find answers for the above stated questions:



It is assumed, that the user who intends to sign the document on the server, will be the owner of a certificate (given by some Certification Center), thus owning a unique user id, a public key and a private key. If he owns one, his id and public key should be stored in so called repository, and held by this Center. The server should have access to this repository, since the data stored there is intended to be public and known to everyone.

To continue with explaining the routines, important details from the chart have to be briefly described, and so, having:

- D - the document being signed
- #D - hash of the document (marked blue)
- $#D - encrypted hash of the document (marked green)
- id - unique user id (marked brown)

The whole routine consists of two major phases, namely, signing the document by the user (above), and verifying authenticity by the server (below). The signing phase looks as follows:

The user chooses a document he wishes to sign, specifies his private key and id, and the ActiveX control located on the website makes the rest, namely, it produces a hash of this document, encrypts it using user's private key, and combines a 'package', consisting of the document, the encrypted hash, and the id. Then, the 'package' is being sent to the server for verification:

Once it is received, it is being decomposed, and three important comparisons have to be made, to assure that the user is really the owner of a certificate and if the document has been received with no changes applied to it on the way. At first, the document from the server is being compared with the one received and decomposed. If they do not differ, the first step has been completed, and we know that the document has not been modified on the way. Then, the decomposed id is being searched for in the repository, to obtain the public key of the user. If it is not found, it means, that the user does not own a certificate. Finally, having obtained the public key from the repository, last part of the 'package', namely the encrypted hash of the document, will be decrypted using this public key. It means that after decoding we have obtained the hash of the document. Since we may produce another hash from the document, it is possible to compare the two afterwards. If they are identical, it means, that the document is authentic and the user is really the one who he intends to be.

Although this phase may seem to be quite confusing, it surely answers the questions stated above and guarantees cohesion of the system.

8.2 Document format vs. XML standard

As it was previously described, the document 'package', which is being sent for verification, consists of the following:

1. The document itself
2. Encrypted hash of the document
3. Unique user id

These three parts of the 'package' were carefully chosen, and are the only necessary for the verification routine. However, the system may be adapted to a new standard, called XML, in the future[5]. The XML format has been designed for attaching signature to HTML and related standards. Such signature consists of several fields, most important of which are:

- Signature id - id of the signature, for classification purposes
- Canonicalization method - an algorithm used to canonicalize the data before digesting it
- Signature method - an algorithm used to encrypt the canonicalized data

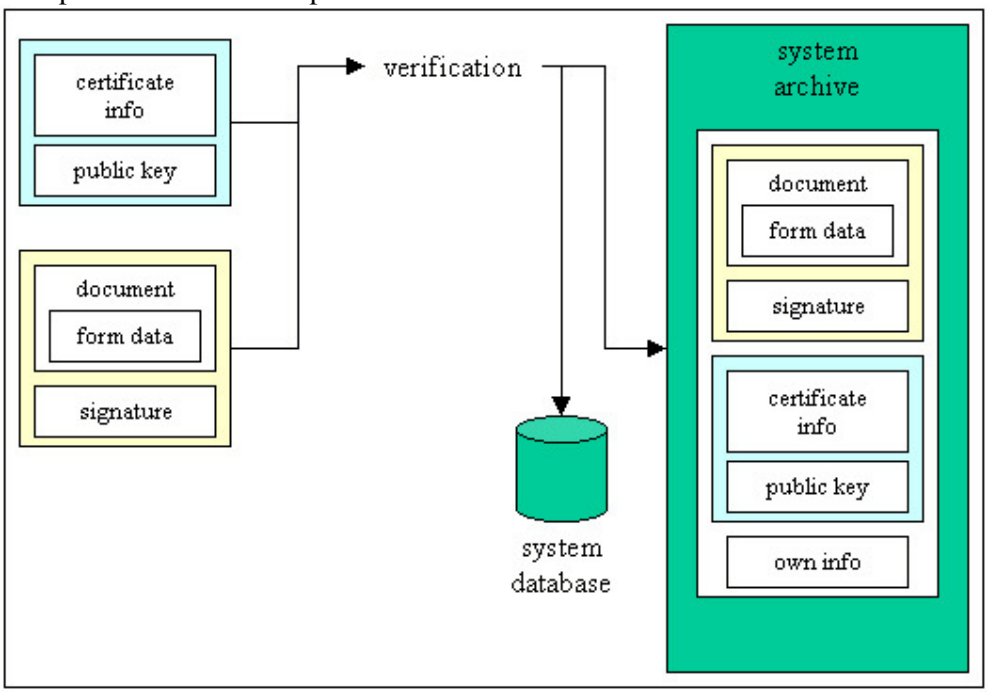
[5] XML-Signature definition – see [10]

- Reference - keeps information about the digest method and the digest value
- Transform algorithm - optional list of steps, taken during the signing process, such as canonicalization, encryption, decryption, validation, etc.
- Digest method algorithm - describes method used during digesting
- Digest value - holds value of the digest
- Signature value - generated value depending on the field structure

The XML standard has been professionally designed by the World Wide Web Consortium (W3C), and supports all necessary options necessary for adding a signature to a HTML document. Since most of them were unnecessary for the project, much more simpler format was used. However, it may be adapted to the XML format, if it is needed in the newer versions of the software.

9. Archiving *(Michał Poręba)*

Due to the nature of the digital signature, it is required to store not only the data entered by the user, but an entire and unchanged document that he signed and submitted. Therefore, after the document had been verified and its validity is confirmed, it is moved to the archive. If any additional information will be needed to be stored along with the document (for example the IP of the terminal he was connecting from), the entire signed document can be nested in a larger XML document along with this information. Additionally, the information about the fact that a specific user filled a specific form is saved in the database.



If, during later system expansion, any information originating from the document is going to be stored in the main database (which would be a sensible option since database searching and other operations will inevitably be faster than extracting the data form archived documents every time it is needed), this data can now be copied into appropriate database records. Alternatively, two databases may be employed –

first one for managing user data, login, session authorization etc, and second – for holding the data that was submitted as a content of the e-forms.

V. Summary *(Michał Poręba)*

1. Summary of the Work

The core system was designed according to the results of the analysis presented in the first part of the thesis, with the scope as defined in the first chapter of the Project Description part of this thesis. Implementation was successfully completed; the system had been installed, configured, and tested on the test machine in the Institute of Computer Graphics.
The next chapter presents a few possibilities of expanding and tuning the system to the needs of possible users.

2. Expansion and Tuning Possibilities

2.1 Client Software

To increase the versatility of the system, clients for other platforms may be written. This includes plugins for web browsers different than MS IE (Mozilla, Netscape, Opera) and software for operating systems different than MS Windows™ - notably Unix/Linux family and MacOS. Possibly Java would be a preferred language to program the parts that cannot be installed as a web browser plugin. The software may be modified and new versions may be introduced whenever such need arises

2.2 Server System

System Integration

The entire system may be incorporated into a larger web application, not loosing its integrity but allowing for more functionality. For example, the system could be a part of a local government portal containing local news, area or city presentation, web directory, weather forecast, company advertisements etc.

System Maintenance

While in its current state the system may be controlled and maintained without major problems from the level of database and by adding new files to the directory, a fully-operational web-administration panel can be prepared, using the same technology as employed in other parts of the system – namely PHP with XHTML and database access. A front-end module for manipulating the data stored in user database can be added in the same way.

Database Engine

A database engine can be upgraded or changed if it is required to do so. This would require some modifications in the PHP code, but another database system could be introduced instead of the MySQL employed.

Database Structure

The database structure can be expanded in order to store more information and fulfill additional requirements that can appear during the system usage. Depending on the scale of the system, new functionality may be added in the database itself, in the server-side scripting or both.

Bibliography

[1] *Dziennik Ustaw Rzeczypospolitej Polskiej*, Nr 130, 15.11.2001

[2] *Extensible Markup Language (XML) 1.0 (Second Edition)*, World Wide Web Consortium, 2002

[3] *HTML 4.01 Specification*, World Wide Web Consortium, 1999

[4] *HTTP Authentication: Basic and Digest Access Authentication*, RFC 2617

[5] *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616

[6] *Microsoft Developers Network* library

[7] *PDF Reference Fourth Edition: Adobe Portable Document Format version 1.5*, Adobe Systems, Inc., 2003

[8] *Rich Text Format (RTF) Specification, version 1.6*, Microsoft Corporation, 1999

[9] *XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition)*, World Wide Web Consortium, 2002

[10] *XML-Signature Syntax and Processing*, World Wide Web Consortium, 2002

[11] C. Adams, S. Lloyd: *Understanding Public-Key Infrastructure*, Macmillan Technical Publishing, 2001

[12] Chris Anley: *Advanced SQL Injection In SQL Server Applications*, Next Generation Security Software Ltd., 2002

[13] B. Kaliski, M. Robshaw: *The Secure Use of RSA*, 1995

[14] Stig Sæther Bakken, Egon Schimd: *PHP Manual*, PHP Documentation Group, 2003