THE CPAL-ES PROTOCOL ANALYSIS ENVIRONMENT

by

Michael P. Runy

A project document submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Florida State University

April, 2001

Approved by _____
Chairperson of Supervisory Committee

_____

_____

_____

Program Authorized
to Offer Degree _____

Date _____

Florida State University

Abstract

THE CPAL-ES PROTOCOL ANALYSIS ENVIRONMENT

by Michael P. Runy

Chairperson of the Supervisory Committee: Professor Alec Yasinsac
Department of Computer Science

In this paper, we show how the Cryptographic Protocol Analysis Language Evaluation System (CPAL-ES) is used to effectively analyze cryptographic protocols in a Windows NT environment. An editor and software library were developed to allow for the integration of CPAL-ES into a homogenous development and analysis environment. Additionally, CPAL-ES was updated and moved from an MS-DOS environment to a Windows NT environment. All of the tools mentioned were developed in order to allow future members of the analysis team to effectively work with the CPAL-ES. We demonstrate how the editor for CPAL was constructed and how it functions. Additionally, we show how the software library is maintained and how it functions. Finally, we show how CPAL-ES was converted from an MS-DOS environment to a Windows NT environment.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to thank Dr. Yasinsac for putting up with me for so long.

1. INTRODUCTION

The objective of this project is to create a homogeneous development environment for the analysis of cryptographic protocols. It is the first step in a workbench of tools that will be developed in order to aid in the timely and accurate analysis of these protocols. These tools are being brought together in a way which will facilitate future users' detailed analysis of protocols and the search for new protocols.

These tools include a syntax-directed editor, and a front-end program, both of which were developed to enhance the functionality of the CPAL Evaluation System (CPAL-ES). CPAL-ES was developed by Dr. Alec Yasinsac in the early 1990's to evaluate and analyze cryptographic protocols [Yasinsac]. The existing CPAL-ES code, written by Dr. Yasinsac was translated from C to C++ and was changed from an MS-DOS based program to a Windows-based program. The editor and front-end were written to make CPAL-ES easier to use and more productive. Additionally, a piece of open source, public domain software was obtained and set up to create a software library where version control and access to protocol code and source code could be controlled. The full process for evaluation of protocols can be seen in the figure below.

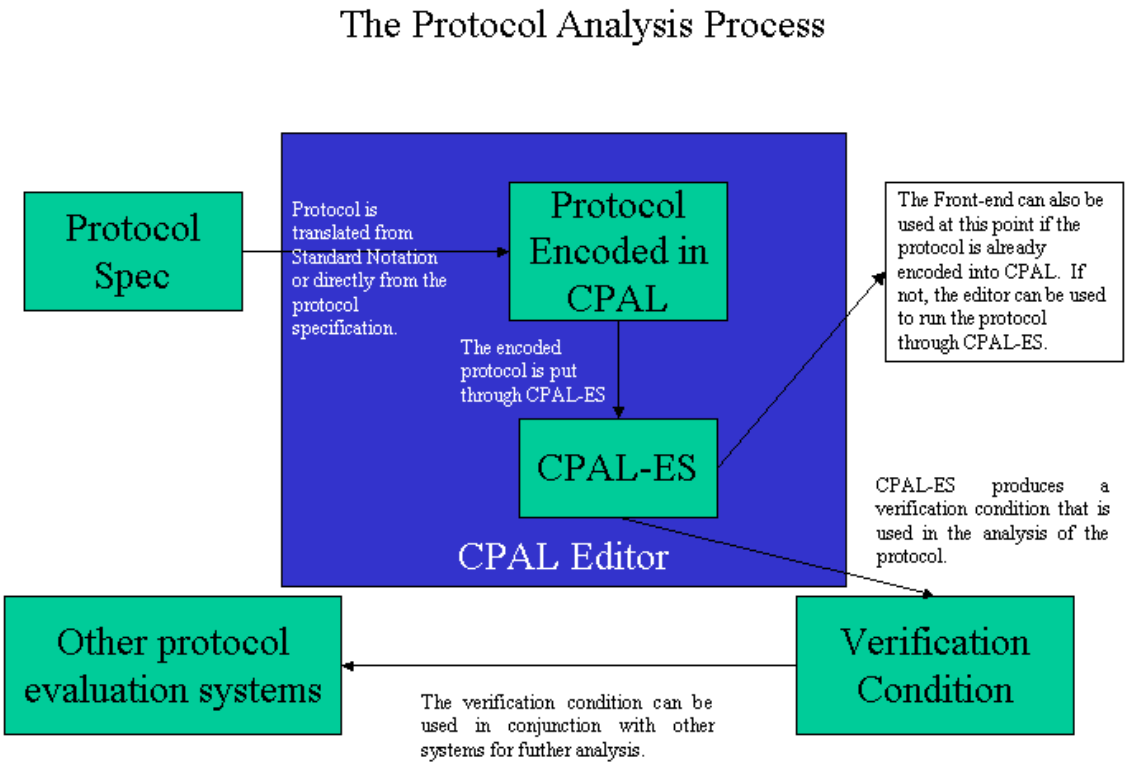

Figure 1

The analysis of cryptographic protocols is a field which is growing in importance by the day. The widespread use of networking has made necessary a method of securing communications over public channels. Cryptographic protocols were developed in order to help accomplish this. The fact that these protocols are supposed to secure communications means that we have to make sure that the protocols themselves are sound and not vulnerable to attack.

Network security is a growing and increasingly important field. Because of the large number of networks currently in existence, and the need for secure communications and transactions over these networks, reliable network security is becoming increasingly necessary. Secure communications are used in a variety of different situations and applications. Ensuring that these communications are secure is the job of protocols that have been designed expressly for this purpose. These protocols have been developed through years of research and there are many different protocols that have been used or proposed for secure communications.

The fundamental technology that allows these protocols to protect our networks is encryption. [Walker] Reliable encryption is essential to these protocols and this encryption can be done in several different levels, such as Link-level, Network-level, and Transport-level encryption. [Nessett] The level of encryption used depends on how much of the information transmitted you wish to keep secret and how much overhead you are willing to deal with to encrypt your information.

There are several different types of threats to network security. Passive attacks are those that cause unauthorized release of information; active attacks are those that cause unauthorized information modification or denial of service. [Voydock] Because of the fact that network security can be compromised in this manner, we cannot assume that any given network environment is benign. Anyone can pretend to be someone else in order to fool people into giving them session keys or other vital information. Because of

this, it is necessary to authenticate that the person you think that you are talking to is indeed who they say they are.

Encryption may be used for authentication and many of the network security protocols that exist today have the concept of authentication built into them. It is an important duty of many protocols to authenticate who is sending and receiving information in order to ensure that the information is being sent to the correct party and not an intruder who may know the protocol steps.

One example of a network security protocol is the Needham and Schroeder protocol. In this protocol, the principals (A and B) contact an authentication server (AS) and, using nonces I1 and I2, are given a secure key (CK) for communication by the server. Nonces are unique numbers that are only used once and are not repeated through the course of the protocol. With ka and kb being shared keys between A and AS and B and AS respectively, the steps of the protocol are:

1. A->AS: B, I_1
2. AS->A: ka[I_1,B,CK,kb[CK,A]]
3. A->B: kb[CK,A]
4. B->A: CK[I_2]
5. A->B: CK[I_2 + 1]

Figure 2

ka[x] denotes encryption of x using key ka

In this example, I_1 and I_2 are nonces that have been generated for this particular protocol run. Nonces are usually random numbers that are generated mainly for the purpose of having a unique identifier for the protocol run. In the first step, principal A sends a message to the authentication server (AS) requesting a session key for use with principal B. The nonce I_1 is generated solely for this run of the protocol and will be unique from other nonces that have been generated in past protocol runs. The authentication server sends back to A a session key (CK) along with a package to send to B. A then passes this along to B who exchanges a nonce (I_2) with A to make sure that the key is fresh and is not an old key from a previous session. There are many other security protocols besides Needham and Schroeder (such as the Otway and Rees protocol [Otway]). These protocols are used for a wide variety of things such as key distribution, digital signatures, authentication, etc.

Because protocols are simply algorithms for interaction between principals, they are subject to the same kinds of flaws to which other algorithms are subject to. For example, Denning and Saco found this flaw in the Needham-Schroeder protocol:

4'. C->B: kb[CK,A]

5'. B->A: CK[I]

6'. C->B: CK[I + 1]

Figure 3

In this attack, a third party has intercepted the first three messages of the Needham-Schroeder protocol and can then pretend to be someone they are not by communicating with one of the other principals using an old key. In this example, the intruder, C, has intercepted A's requests to the authentication server and then uses an old, compromised key to communicate with B. The intruder may or may not know what the contents of kb[CK,A] are but they know that the step in the protocol tells them that this is used to establish a session with principal B. Principal B has no way of knowing that the person that they are communicating with is actually an intruder. Principal B assumes that the key that C has given to them is valid, when in fact, it is an old key from a previous session that has been compromised.

Protocol verification is an essential part of the analysis of cryptographic protocols. We must know that the protocols are acting in the method that they are intended to act. Finding weaknesses such as the Denning and Saco flaw is a big part of this. Additionally, protocols must impart the necessary information to the principals without compromising the security of the communication. For example, protocols should be able to deliver the session keys to both principals involved in a communication without allowing those keys to be compromised.

Logics

Several logics were developed as a formal way to verify that protocols met their goals. These logics represented a paradigm shift in the way that protocols were looked at because they introduced the idea of belief in addition to actions and data. The first logic

was developed by Burrows, Abadi, and Needham and is called BAN logic. [Burrows] BAN logic provided a formal way to describe an authentication protocol. In addition to the formal notation that the authors developed, several types of logical postulates were also developed. For example, the nonce verification rule states that if principal P believes that nonce N is fresh and that principal Q once said X, then P will believe that Q believes X. This notion of belief in dealing with cryptographic protocols was new. It allowed for a more in-depth analysis of how principals related to each other in protocols and how their actions would affect each other down the line.

Others expanded on the logics that Burrows, Abadi, and Needham developed by adding more logical operators and extending the logics further. The GNY [Gong] logic, for example, added the idea of implication to the logic. It provided a way to distinguish between the content of a message and the information implied by the message. [Gong] Additionally, logical operators were added, such as, the unless operator added by Moser. [Moser]

Weakest Precondition Reasoning

Weakest precondition reasoning was first used by Yasinsac in the evaluation of protocols [Yasinsac]. Weakest precondition reasoning allows us to take a set of postconditions that we would like to result from a protocol run and find out what the weakest set of preconditions for those postconditions would be. It takes the actions of a protocol into account as it is determining what the weakest preconditions for the given postcondition are.

For example, P{S}Q is a logical statement, which can be evaluated as either true or false. In this case, P is a set of preconditions, S is a protocol, and Q is the desired outcome expanded as a set of postconditions. So, if P is true when S begins executing, then Q will be true after S is finished. The objective, then, is to find a precondition, P, that when run through S will result in Q being true.

Suppose we have a program statement, y := x/k and a desired postcondition of x = y. The weakest precondition for this statement would be the statement that insures that x would be equal to y after the program segment executes. In this simple case, it is easy to see that in order for this to be true, k must equal one. So, given our notation, the entire statement can be expressed as: k = 1{y := x/k}x = y.

In the realm of protocol analysis, weakest preconditions are used to find a desired precondition that will verify the correctness of a protocol. This is called the verification condition. This verification condition takes the form of a logical predicate that can be simplified to TRUE if a given protocol correctly executes. This is done using predicate calculus and using logical axioms in order to simplify the predicate.

Preconditions are formed from protocol assumptions, assertions, and actions. For example, in order to properly verify a protocol, we might have to assume that principal A holds the same session key as principal B. Actions can also represent a valid precondition. Actions such as retrieving a key from a key server, or making sure that a nonce is fresh by verifying it with the other party to the communication, may be necessary to a successful run of the protocol.

The CPAL-ES system produces a verification condition for a given protocol. This verification condition is then simplified in an attempt to make the predicate easier to

understand. By evaluating protocols using weakest precondition reasoning, we seek to find flaws that exist in these protocols. Additionally, if no new flaws are found, a more thorough understanding of the workings of the protocol is gained.

Other Analysis Tools

There are several other analysis methods that can be used to evaluate protocols, including CSP, the Interrogator [Millen], and the Naval Research Laboratory Protocol Analyzer (NRL PA) [Meadows]. All of these tools have been developed in order to allow for the automatic verification and analysis of protocols. Instead of having to do this verification by hand, these tools allow us to evaluate protocols much more quickly, which, in turn, allows us to evaluate the more complex protocols that are being developed every day.

The NRL PA was developed by the Naval Research Laboratory for the analysis of cryptographic protocols that are used to authenticate principals and services and distribute keys in a network. It is programmed in Prolog and uses many features of that language to speed up the development cycle and allow for rapid changes to the program if necessary. The basic technique used by the NRL PA is one that uses an insecure state as its start state (it is actually a final state) and then tries to find out if that state is reachable by the protocol. Unreachable states are discounted and the number of protocol runs can be unlimited, allowing for an in depth analysis of the protocol.

In addition, Artificial Intelligence (AI) can be used to evaluate protocols. The Interrogator is one example of a tool that uses AI for this type of analysis. [Millen] It is programmed in Prolog and uses the AI capabilities of Prolog to narrow its search for weaknesses in the protocol. It can be directed by the user of the program to search for specific weaknesses or flaws. It is a goal-directed program, which means that it will avoid futile paths that cannot be taken by an intruder.

The primary tool with which we are concerned is CPAL-ES. The rest of this paper will show how an integrated workbench is being developed which allows for easier use of CPAL-ES. First, the development of a software library, which allows for version control and parallel development of code will be discussed. Second, the conversion of the CPAL-ES code from MS-DOS based to Windows NT based code will be covered, including the development of a front-end program designed to make execution of the CPAL-ES easier. Finally, the development of a syntax-directed editor for CPAL will be discussed along with further goals for the workbench. These steps comprise the majority of the work involved in creating a homogeneous environment in which the evaluation of protocols would be both easier and quicker.

2. The Security Group Software Library

The first focus of developing the integrated workbench was to find a solution for version control of the code that was being developed. A software library in which various versions of the code being developed could be stored was the desired outcome. This library would keep track of the different versions of code that it contained. The tool would also be able to manage protocol versions so that we could improve protocols and then verify the improvements. Additionally, it would enable members of the security group to check out pieces of code in order to update or revise them. In doing so, the tool would ideally keep track of who made what changes, and which version was the most current version of the code. The platform for the software was to be Windows NT.

The best solution obtained was via a piece of public domain software named WinCVS. This software is an open source program that was developed by a group looking for a Windows front-end to the popular UNIX CVS program. WinCVS contains many of the features that we desired for our version control system. Other software such as QVCS and JCVS were evaluated but they were deemed ineffective for the job that we had in mind. WinCVS presented the best options and was actually designed to mix with other operating systems such as UNIX and the MacOS in addition to a Windows environment.

WinCVS is a menu-driven configuration management system. The first step in using WinCVS is creating a “module”. This module contains all of the code which the user wants protected and monitored. The directory hierarchy that the user wants to be

part of the library is imported into the CVS server. Once there, it can be checked out, viewed, or archived. A password file maintained on the server determines who can or cannot access the files that are being protected by WinCVS.

The figure below shows a sample main window and what it looks like when populated with a module.

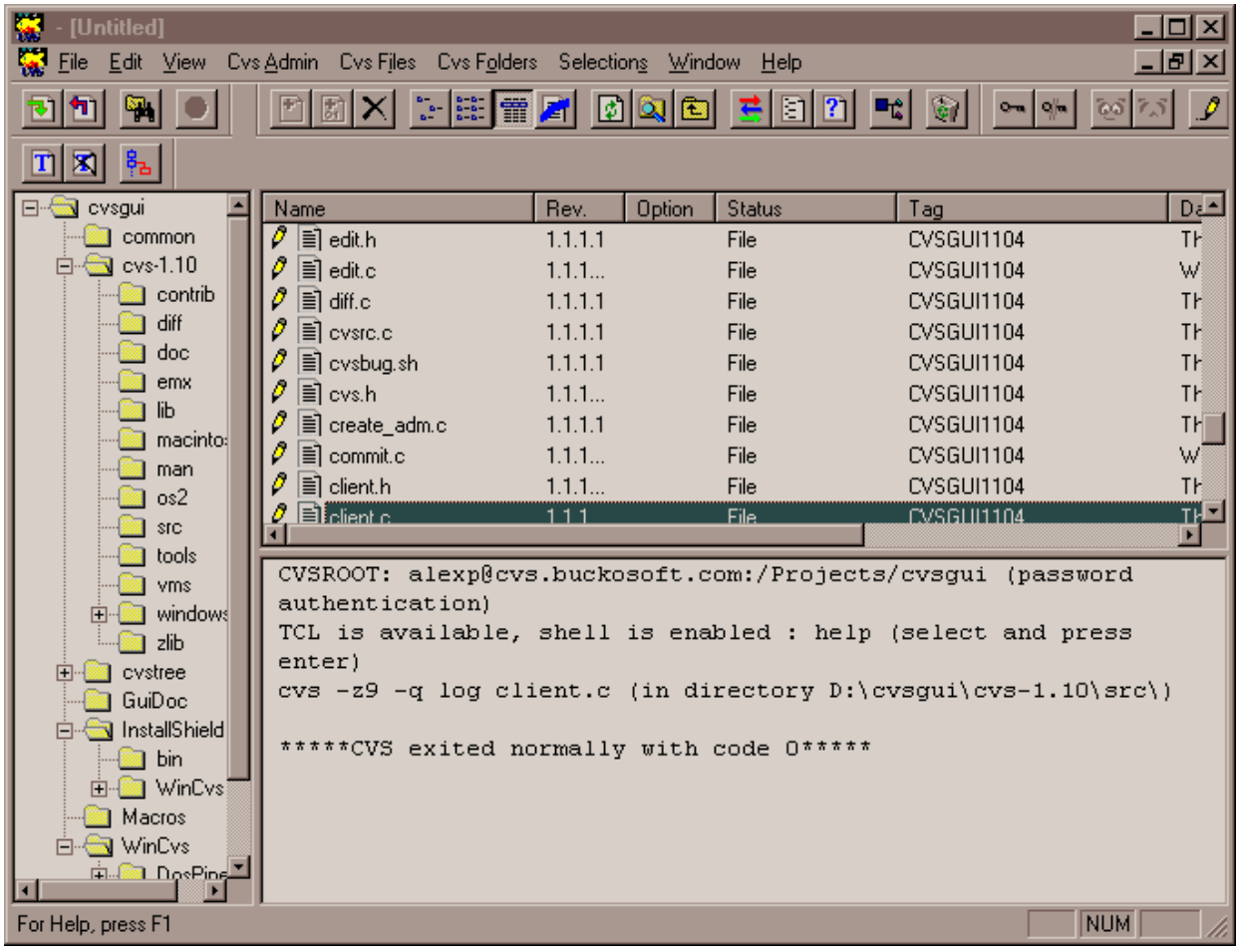

Figure 4

The buttons at the top of the screen allow the user to check out or check in files, lock them so no one else can use them, add labels to them, and compare them with previous versions of the file in addition to several other operations. The main window of

the program shows the directory structure of the server on the left side, along with the files that are contained within the currently selected project on the right-hand side. Below the file list is a window that contains debug information on particular commands, as well as a command line where CVS commands can be entered without using the menu options. Additionally, information is displayed in the bottom window when revision histories and file information is asked for.

The WinCVS software allows us to control access to source code for the various components of the security workbench. Anyone can create a module for code that they are working on and allow whomever they want to have access to those files. Additionally, older revisions of code are stored so that they can be accessed at a later date if necessary. A wide variety of reports can also be created that show how revisions have been made as well as who made them. Command-line utilities are also provided that allow for automation of some configuration management tasks.

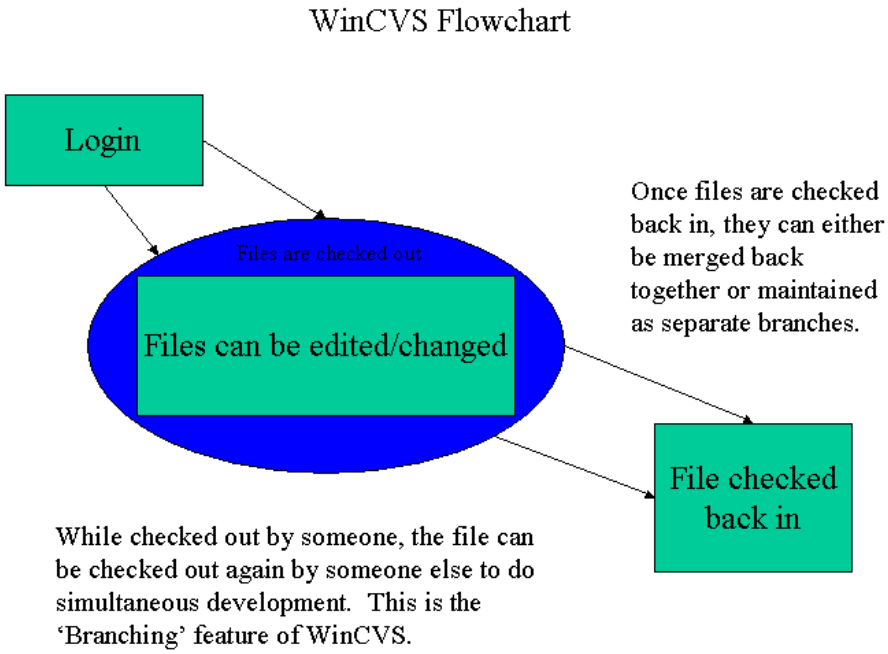

Figure 5

As an added bonus, WinCVS allows for what it calls "branching". Branching is where several different versions of code can be simultaneously developed along different "branches". Whenever the code is in working order, these branches can be merged back together into a single piece of code. The "branches" can also be maintained independently for an indefinite period of time.

The main obstacle to setting up the software library was becoming familiar with the software and how to install it properly. For instance, the software uses a password file that must be located within the root directory of the install. However, it only recognizes an encrypted password, much like a Unix password file. The encrypted passwords necessitated the use of a separate piece of software that allowed for the

creation of the password file. Additionally, the default install directory was not used for our library since a more central location that existed on a shared drive was desired.

The login process by the user was another one of the problems faced in implementing this piece of software. The password file was one issue but another was how the person checking out the files would actually log in to allow them to do so. This is done from the main screen of WinCVS. In the 'CVSAdmin' menu, there is a Login selection. This was originally thought to apply only to administrators of the software library, but is in fact, the general login that all users would use. The documentation and upgrades to the software are located at http://www.wincvs.org.

3. CPAL-ES: From DOS to Windows

The second focus of the development environment was to port CPAL-ES to Windows NT from an MS-DOS environment. This consisted of several steps, beginning with the initial conversion to Windows-based code, conversion from that code into C++ code, and finally, the development of a Windows-based front-end for the CPAL Evaluation System.

The conversion of the code into usable code that could be compiled and executed within a Windows environment was the first step in getting the workbench up and running. The first step in this process involved creating a project and workspace within the Microsoft Visual C++ development environment for the CPAL-ES system. This was accomplished without major problems. The next step consisted of testing the program and verifying that it compiled and ran under Windows NT. There were some conversions from C code to C++ code that were done at this step in order to port the program. Things such as array initialization and how to get the program to run in a console window were the main hurdles to obtaining running code.

The code was compiled within the Visual C++ development environment and then tested using several previously coded protocols. However, the code still required an MS-DOS environment to run, albeit a windowed one which operated under Windows NT. It was still command-line driven which was a hindrance to the operation of the system in a Windows environment.

The conversion of the CPAL-ES code from C to C++ was partially done by a member of the Young Scholars Program, Will Gross. He converted most of the functional code to C++ and created classes and methods that operate on those classes while still retaining the functionality of the program. The conversion of the LEX and YACC files, was done by myself along with the conversion of the previous main() function from the legacy C code. There are some conversions that remain to be done including some functions, methods, constructors and destructors but, for the most part, the entire program now exists in C++ form. This includes the parser files that were generated by LEX and YACC given the existing grammar.

However, even after this conversion to C++ code, the program was still a command-line program and not a true Windows application. For this, we elected to develop a front end that would use the existing code but present a GUI that was Windows oriented.

The Windows-based front-end for the code was created from within the Visual C++ development environment. A Microsoft Foundation Class (MFC) based program was developed that utilized the existing CPAL-ES program and allowed the user to input the protocol that they desired to be evaluated in a dialog box. The MFC libraries allow for use of standard Windows functions and portability of the code across various Windows platforms. This dialog box then launches the CPAL-ES application with the given protocol as input.

Using the Dialog classes that are part of the MFC library, the program creates a dialog box that contains an edit box which asks the user for the location of the protocol which they wish to evaluate. Upon entering the name of the file containing the protocol specification, the program then calls the CPAL-ES program with the filename as an argument. The CPAL-ES system still uses the Windows console to display its temporary output and, finally, the files that are created for analysis are opened by the program so that the user can continue their evaluation of the protocol.

The method that allows this to happen is a method called OnOk(). This method is called when the user presses the OK button in the dialog box. It first saves the information in the edit box that the user has input to a CString variable. This is the string class developed for use with MFC applications. This string is then converted to a form that can be passed to the CPAL-ES program. Following this, the ShellExecute() command is used to execute the CPAL-ES program with the parameter specified by the user. When its execution has completed, the console window closes and the three files that have been created by CPAL-ES are opened for evaluation.

```
void CFrontendDlg::OnOK()
{ UpdateData(); // This allows the value in the edit box to be used
// The next line executes the CPAL program with the parameter m_edit1
// which is the file name entered in the dialog box.
ShellExecute(NULL,"open","cpal.exe",m_edit1,NULL,SW_SHOW);
Sleep(500);
ShellExecute(NULL,"open", "protocol.out", NULL, NULL, SW_SHOW);
ShellExecute(NULL,"open", "assume.out", NULL, NULL, SW_SHOW);
ShellExecute(NULL,"open", "tst.out", NULL, NULL, SW_SHOW);
CDialog::OnOK();}
```

Figure 6

Figure 4 shows the OnOk() method. This is the method that executes the CPAL-ES program and also opens the relevant documents after execution. The call to UpdateData() allows the string entered in the edit box to be used as a Cstring and passed to the CPAL-ES program as a parameter. Following the UpdateData() call, the CPAL-ES program is executed via a ShellExecute() command with m_edit1 as a parameter. The value in m_edit1 is the name of the protocol that has been entered by the user. Finally, the three ShellExecute() commands at the end open the files that are output by CPAL-ES.

This front-end program makes the execution of CPAL-ES in a Windows environment a reality. It does not change the functionality of the previously existing code thereby allowing one to use a command line interface if one so desires. However, it does make it easier to use within the Windows context. It may eventually be phased out as the workbench becomes more integrated and more functional but for now, it provides an easy interface with which to interact in order to use the CPAL-ES system.

4. The CPAL Editor

The final step in creating the integrated work environment was the development of a syntax-directed editor for CPAL. This editor highlights syntax much like most Windows-based programming tools do. Additionally, it identifies syntax errors to the person using it and compiles the protocols if asked to do so. The editor was developed in order to facilitate the translation of protocols into CPAL so as to allow people who will translate these protocols to do their job more quickly and efficiently with fewer mistakes.

The CPAL editor was designed as an SDI (Single Document Interface) application with MFC support. The framework for the editor was actually created by Microsoft Visual C++. Additional support was added after the basic framework was already in place to highlight the syntax and allow for compilation of the protocol being edited from within the editor itself.

The document/view architecture of MFC works by associating a particular view with each document. In this case, the CCPALEditorView class implements the functionality of the syntax highlighting and the compilation. CCPALEditorView is a child of the CRichEditView class. This class is one which is usable by all MFC applications in order to implement not only common editing functions such as cut and paste, but also character formatting and other more complicated operations. Both the CRichEditView and the CEditView class deal with operations having to do with text editing. The use of the CRichEditView class was necessary in order to provide coloration to the text.

The document view (e.g. – CCPALEditorView) is, basically, the actual editing window that we are using at the moment. The first thing that is done upon creation of the view is to execute the OnCreate() function of the CCPALEditorView class. This function, listed in Appendix A, first creates a list of keywords that the editor will use for syntax highlighting. It does this through use of the GetTokenTypes() function. GetTokenTypes() retrieves the names of the keywords from a structure that is generated whenever the YACC grammar specifying the language is compiled. In this way, the grammar of the language can be changed without having to change the editor.

The keywords of the language are added to a static list by the AddKeywords() function. This list is then used as a basis of comparison while editing in order to effect the syntax coloring. Nonces and keys are also set up in the OnCreate() function. Basically, any nonce or key will be highlighted accordingly as it is typed in.

The remainder of the work is done throughout the CCPALEditorView class. The FormatTextRange() function is the most important one as it does all of the syntax highlighting as the user types in the protocol. Shown below, the code first creates a buffer and then checks this buffer against the known types of keywords, comments, etc.. Whenever it finds a word that is reserved, it colors it accordingly using the designated color that we determined earlier in the code. It processes comments, nonces, keys, numbers, keywords, and constants all from within the same function.

This FormatTextRange() function is called whenever the text in the view is changed. The range on which the formatting is done is determined by the operation that

is changing the text. When typing in text, the range is just the current cursor position. However, operations for cut and paste, etc. were added to insure that all of the text will remain formatted with the syntax coloring.

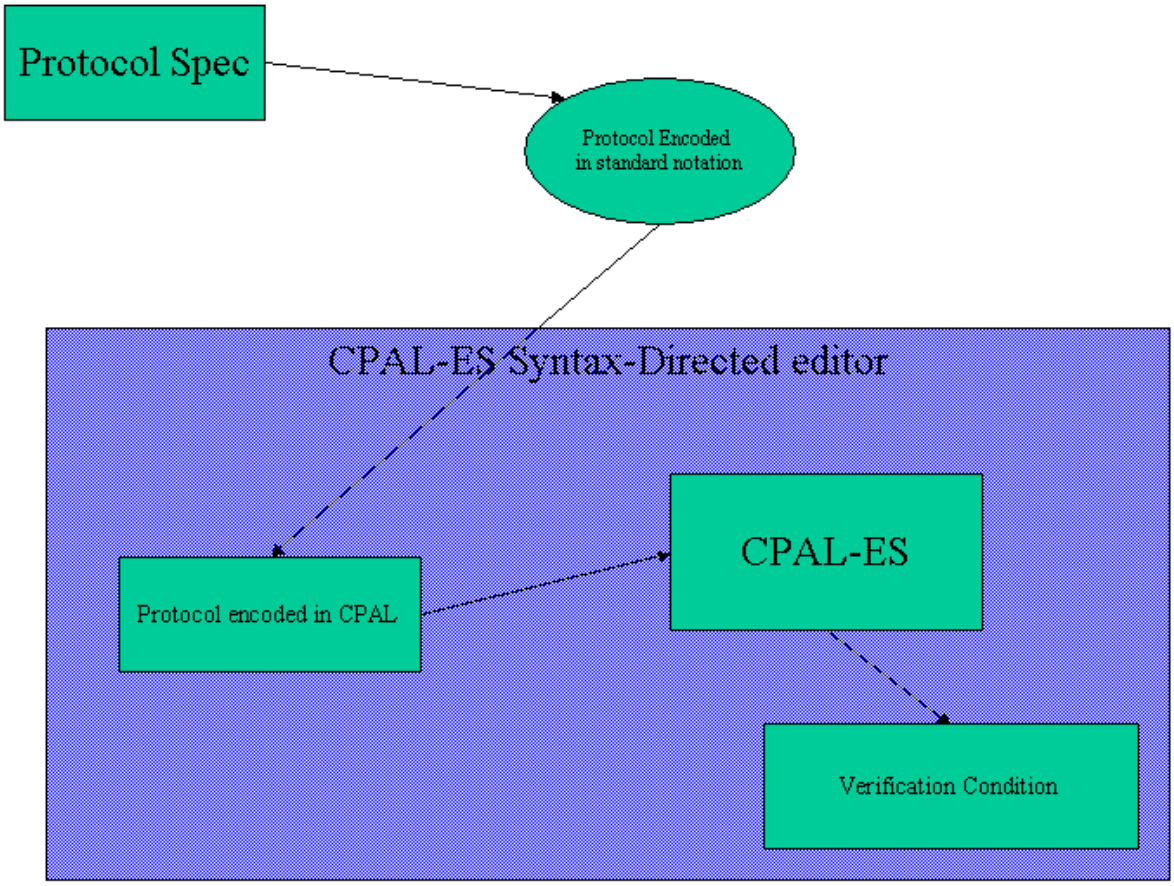

Figure 7

The other main addition to the simple editor view is the addition of a compile button in the toolbar. The button is a simple C (standing for Compile) that was added to the end of the toolbar. When the button is pressed, a message is passed which calls the OnCompile() function of the CCPALEditorView class. This function streams the text out

to a temporary file which is then passed to the parser created by LEX and YACC. Some modification of the standard error function created by LEX and YACC was necessary as well in order to have message boxes pop up which detail any syntax errors encountered in the code. Following successful compilation, the three output files created by CPAL-ES are opened for analysis. The code used to call the parser and implement the CPAL-ES are mostly unchanged from the original implementation of the program.

The files are all closed after their use which allows the ShellExecute() command to open the three documents that are created by the CPAL-ES. Additionally, the temporary file that was created in order to use as input to the parser is destroyed as it merely contains a text representation of what is already contained in the editor.

5. Conclusion

The widespread development and use of cryptographic protocols for security has brought with it a necessity to analyze these protocols, not only to find flaws, but also to be able to understand the protocols and how they work more effectively. The use of logics and formal methods in this field were developed expressly to do just that. These logics are combined with weakest precondition reasoning in the CPAL-ES.

The development of a full-scale workbench for use with CPAL-ES is just beginning. These are only the first few steps in a process which will eventually lead to an integrated tool that enables users to easily enter protocol specifications in CPAL, run them through CPAL-ES, and analyze their results. This workbench will facilitate the development of more secure network protocols by exposing flaws that may be discovered or just by allowing a more in-depth understanding of what makes a good or bad protocol.

The tools that have been created so far: The CPAL Editor, and Front-end combined with the software engineering environment supplied by the QVCS program will allow continued development of this workbench. They are the first components in what will eventually be a much more robust tool.

BIBLIOGRAPHY

[Burrows] Burrows, Michael, Abadi, Martin, and Needham, Roger. "Authentication: A Practical Study in Belief and Action," Digital Equipment Corp. Systems Research Center.

[Denning] Denning, Dorothy E. and Sacco, Giovanni Maria. "Timestamps in Key Distribution Protocols," *Communications of the ACM*, Vol. 24, No. 8, August 1981.

[Gong] Gong, Li, Needham, Roger, and Yahalom, Raphael. "Reasoning about Belief in Cryptographic Protocols," in *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, 1990.

[Meadows] Meadows, Catherine. "The NRL Protocol Analyzer: An Overview," *Journal of Logic Programming*, Vol. 26, No. 2, February 1996.

[Millen] Millen, Jonathan K., Clark, Sidney C., and Freedman, Sheryl B. "The Interrogator: Protocol Security Analysis," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 2, February 1987.

[Moser] Moser, Louise E. "A Logic of Knowledge and Belief for Reasoning about Computer Security," Department of Computer Science University of California.

[Needham] Needham, Roger M. and Schroeder, Michael D. "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, Vol. 21, No. 12, December 1978.

[Nessett] Nessett, D. M. "Factors Affecting Distributed System Security," Lawrence Livermore National Laboratory, 1986.

[Otway] Otway, Dave, and Rees, Owen. "Efficient and Timely Mutual Authentication," *Operating Systems Review*, Vol. 21, No. 1, January 1987, Pgs. 8-10

[Voydock] Voydock, Victor L. and Kent, Stephen T. "Security Mechanisms in High-Level Network Protocols," *Computing Surveys*, Vol. 15, No.2, June 1983.

[Walker] Walker, Stephen T. "Network Security: The Parts of The Sum," Trusted Information Systems, Inc., 1989.

[Yasinsac] Yasinsac, Alec. "A Formal Semantics for Evaluating Cryptographic Protocols," Ph.D. Dissertation, University of Virginia, January 1996.

APPENDIX A (Code Samples)

The following function is the OnCreate() function. This details what happens when the main editor window is created.

```
int CCPALEditorView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{ int index = 256;
  CString Keywords(" ");
  while( (index >= 256) && (index <= 304) )
  { Keywords += GetTokenTypes(index);
    Keywords += " ";
    index++;
    AddKeywords(Keywords);
  }
  if (CRichEditView::OnCreate(lpCreateStruct) == -1)
    return -1;
  Initialize();
  SetCaseSensitive(FALSE);
  SetNonce(_T("N"));
  SetKey(_T("k"));
  AddKeywords(szKeywords);
  AddConstants(szConstants);
  return 0;
}
```

The FormatTextRange() function is the function that goes through the text in the editor window and formats its color, etc..

```
void CCPALEditorView::FormatTextRange(int nStart, int nEnd)
{if (nStart >= nEnd)
  return;
  m_bInForcedChange = TRUE;
  CHARRANGE crOldSel;
  GetRichEditCtrl().GetSel(crOldSel);
  LockWindowUpdate();
  GetRichEditCtrl().HideSelection(TRUE, FALSE);
  TCHAR *pBuffer = NULL;
  try { GetRichEditCtrl().SetSel(nStart, nEnd);
    pBuffer = new TCHAR[nEnd - nStart + 1];
    long nLen = GetRichEditCtrl().GetSelText(pBuffer);
    ASSERT(nLen <= nEnd - nStart);
    pBuffer[nLen] = 0;
    TCHAR *pStart, *pPtr;
    pStart = pPtr = pBuffer;
    TCHAR* pSymbolStart = NULL;
    SymbolColor ic;
    while (*pPtr != 0)
     {  TCHAR ch = *pPtr;
      if (ch == '-')
      { pSymbolStart = pPtr;
        ch = *(++pPtr);
        if (ch == '-') {
          do {
            ch = *(++pPtr);
          } while (ch != 0 && ch != '\r');
        ic = m_icComment;
        }
      } else if (IsNonce(ch)) { // Process Nonces
        pSymbolStart = pPtr;
        TCHAR ch1 = ch;
        do {
          ch = *(++pPtr);
        } while (ch != 0 && ch != ch1 && ch != '\r' && (_istalpha(ch) || ch == '\'') );
        if (ch == ch1) pPtr++;
        ic = m_icNonce;
      } else if (IsKey(ch)) { // Process Keys
```

```
        pSymbolStart = pPtr;
        TCHAR ch1 = ch;
        do {
          ch = *(++pPtr);
        } while (ch != 0 && ch != ch1 && ch != '\r' && (_istalpha(ch) || ch == '\'') );
        if (ch == ch1) pPtr++;
        ic = m_icKey;
      } else if (_istdigit(ch)) { // Process numbers
        pSymbolStart = pPtr;
        _tcstod(pSymbolStart, &pPtr);
        ic = m_icNumber;
      } else if (_istalpha(ch) || ch == '_') { // Process keywords
        pSymbolStart = pPtr;
        do {
          ch = *(++pPtr);
        } while (_istalnum(ch) || ch == '_');
        *pPtr = 0;
        int nPos = IsKeyword(pSymbolStart);
        if (nPos >= 0) {
ChangeCase(nStart + pSymbolStart - pBuffer, nStart + pPtr - pBuffer, m_strKeywords.Mid(nPos+1, pPtr
- pSymbolStart));
          if (_tcsicmp(m_strComment, pSymbolStart) == 0) {
            *pPtr = ch;
            *pSymbolStart = m_chComment;
            if (pSymbolStart[1] != 0 && m_chComment2 != 0)
                                                          pSymbolStart[1] = m_chComment2;
            pPtr = pSymbolStart;
            pSymbolStart = NULL;
            continue;
          }
          ic = m_icKeyword;
        } else {
  //Process Constants
          nPos = IsConstant(pSymbolStart);
          if (nPos >= 0) {
ChangeCase(nStart + pSymbolStart - pBuffer, nStart + pPtr - pBuffer, m_strConstants.Mid(nPos+1,
pPtr - pSymbolStart));
            ic = m_icConstant;
          } else {
            pSymbolStart = NULL;
          }
        }
```

```
        *pPtr = ch;
      } else {
        pPtr++;
      }
      if (pSymbolStart != NULL) {
        ASSERT(pSymbolStart < pPtr);
SetFormatRange(nStart + pStart - pBuffer, nStart + pSymbolStart - pBuffer, FALSE, RGB(0,0,0));
SetFormatRange(nStart + pSymbolStart - pBuffer, nStart + pPtr - pBuffer, ic.bBold, ic.clrColor);
        pStart = pPtr;
        pSymbolStart = 0;
      } else if (*pPtr == 0)
SetFormatRange(nStart + pStart - pBuffer, nStart + pPtr - pBuffer, FALSE, RGB(0,0,0));
    }

  } catch(...){}
  delete [] pBuffer;
  GetRichEditCtrl().SetSel(crOldSel);
  GetRichEditCtrl().HideSelection(FALSE, FALSE);
  UnlockWindowUpdate();
  m_bInForcedChange = FALSE;
}
```

The OnCompile() function is executed when the compile button is pushed within the editor. It runs the protocol through CPAL-ES.

```
void CCPALEditorView::OnCompile()
{ CRichEditCtrl& r = GetRichEditCtrl();
  CFile fWrite("c:\\test.txt", CFile::modeCreate | CFile::modeWrite);
  EDITSTREAM strm;
  strm.dwCookie = (DWORD) &fWrite;
  strm.pfnCallback = WriteEditData;
  r.StreamOut(SF_TEXT, strm);
  fWrite.Close();

  extern STMT_PTR start,lst_stmt;
  extern PREDICATE Q = NULL;
  extern GIVEN assumptions;
  yyin = fopen("c:\\test.txt","r");

  start_stack();
  msgctr = 0;
  totmem = 0;
  out_ptr = fopen("tst.out","w");
  out_list = fopen("protocol.out","w");
  assume_list = fopen("assume.out","w");
  assumptions = NULL;
  printf("in wp.c, start = %ld\n",(long)start);
  printf("\n-----------------------\n\n");

  if (!yyparse())
  {
    Q = new pred(_BOOL);
    Q->bool = TRUE;
    printf("\nFinished parsing, ready to generate predicate.\n\n");
    Q = start->wp(Q);
    delete start;
    printf("Printing UNSIMPLIFIED predicate\n\n");
    fprintf(out_list,"\n\n\n****** Initial WP predicate follows.\n\n");
    Q->print_pred();
    fprintf(out_list,"\n\n\n****** Simplified predicate follows.\n\n");
    printf("\n\n*** Secondary Simplification \n\n");
    Q = Q->simp();
    fprintf(out_list,"\n\n\n****** Simplified predicate follows.\n\n");
```

```
    printf("\n\n*** PRINTING simplified predicate. \n\n");
    Q->print_pred();
    fprintf(out_list,"\n\n****** NO MORE PREDICATE\n");
    printf("All is finished, totmem = %ld, totfre = %ld.\n",totmem,totfre);
    printf("%d calls to copy_val.\n",tmpctr);
    printf("%d calls to copy_pred.\n",predctr);
  }
  fclose(out_list);
  fclose(yyin);
  fclose(out_ptr);
  fclose(assume_list);
  fWrite.Remove("c:\\test.txt");
  ShellExecute(NULL,"open", "protocol.out", NULL, NULL, SW_SHOW);
  ShellExecute(NULL,"open", "assume.out", NULL, NULL, SW_SHOW);
  ShellExecute(NULL,"open", "tst.out", NULL, NULL, SW_SHOW);
}
```