Automated Attack Analysis and Code Generation in a Multi-Dimensional Security Protocol Engineering Framework

Simon Lukell, Christopher Veldman and Andrew Hutchison
Data Network Architecture Group
Department of Computer Science
University of Cape Town
Rondebosch 7701
Ph: (021) 650 3127
Fax: (021) 689 9465
{slukell, cveldman, hutch}@cs.uct.ac.za

***Abstract*— A unified, multi-dimensional approach to security protocol engineering is effective for creating cryptographic protocols since it encompasses a variety of design, analysis and implementation techniques, thereby providing a higher level of confidence than individual approaches. SPEAR II, the Security Protocol Engineering and Analysis Resource II is a tool which supports this unified, multi-dimensional approach by offering protocol designers an environment in which to graphically design and analyse security protocols. The premise of this paper is that providing a means of analysing security protocols for attack vulnerabilities that are not detected by static methods, and allowing for the translation of an abstract protocol specification into a correct and secure protocol implementation in the context of a multi-dimensional tool such as SPEAR II, will assist in producing more secure security protocols.**

I. Introduction

Security protocol design, analysis and implementation have become so advanced and complex that it is not viable to perform certain moments by hand as they take too long and/or tend to become tedious and error-prone over time. Specialised tool support for formal methods can significantly aid protocol engineers in creating and implementing more secure cryptographic protocols, thus helping to prevent errors that often creep into protocol implementations.

Each of the techniques currently available to the security community is not capable of detecting every possible flaw or attack against a protocol when used in isolation. However, when used in combination with other formal methods, they all complement each other and allow a protocol engineer to obtain a more accurate overview of the security of a protocol which is being designed. What is required is a unified approach to protocol engineering, one which combines a number of protocol engineering dimensions into one application that is consistent and easy to use. *Multi-dimensional security protocol engineering* is an effective approach for creating and deploying cryptographic protocols, since it encompasses a variety of analysis techniques, thereby providing a higher security confidence than individual approaches can achieve [1].

A tool that employs such a multi-dimensional engineering approach is the Security Protocol Engineering and Analysis Resource (SPEAR) II [2]. SPEAR II is a tool that adopts a unified, multi-dimensional approach to security protocol engineering and analysis. It consists of various interacting modules, combined into one graphical user interface, which equip protocol engineers with an easily accessible array of proven techniques with which they can design and analyse a security protocol in an efficient and controlled manner.

II. The SPEAR II Framework

An overview of the SPEAR II Framework is shown in Figure 1. The modules employing the attack analysis and code generation techniques presented in this paper are coloured in black in the figure. The figure shows how the protocol specification module of SPEAR II called GYPSIE [3] serves as a basis for all the other dimensions. Protocol specification in SPEAR II is done graphically in the GYPSIE environment, while GNY belief logic [4] analysis is done using the Visual GNY and GYNGER components of the SPEAR II environment.

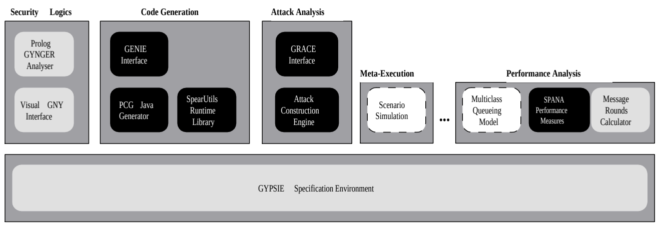

Fig. 1. Scope and ambitions of the SPEAR II Framework.

GYPSIE protocol specifications which are only subjected to belief logic analysis such as GNY can still contain certain flaws. In order to increase the confidence of protocols specified in SPEAR II, the attack analysis dimension in Figure 1 was investigated and implemented. The aim of this module is to analyse a GYPSIE protocol specification for security vulnerabilities and detect certain flaws that GNY logic analysis is incapable of finding. The implementation includes a graphical attack analysis environment called GRACE and a dynamic analysis engine called ACE. Both of these components are shown in the figure, and are described in Sections III and IV.

In order for the abstract specification and logical analysis of security protocols to be beneficial to the security protocol engineering process, methods are required which provide an efficient and effective translation from the high level protocol abstraction to a secure and correct low level implementation. Therefore, the aim of security protocol code generation in the context of the multi-dimensional SPEAR II environment is to provide a translation from the protocol specification in GYPSIE and any associated protocol analysis information, into a secure protocol implementation. The addition of a code generation module to the SPEAR II Framework also allows for the inclusion of performance analysis methods into the SPEAR II Framework. Automatic code generation of security protocols specified in SPEAR II is implemented via the graphical code generation environment called GENIE and the protocol code generator PCG, which are described in Sections V and VI.

Another important aspect in protocol engineering is the performance of the implementation. To assist the engineer in evaluating the generated code, a performance analyser module called SPANA has been incorporated into the SPEAR II Framwork. This module enables measurement of various performance metrics of different cryptographic libraries and algorithms and is described in Section VII.

III. The Attack Construction Engine

The core of the attack analysis module is the **A**ttack **C**onstruction **E**ngine (ACE). One of the main features of ACE is that it is capable of modelling *constructed keys*, *i.e.* the attacker in the model can use more than one component, possibly encrypted, as a key to send and receive encrypted messages. ACE is based entirely on the method developed in [5]. This model converts the reachability problem (finding a state that violates some condition), into a constraint solving problem, which can be analysed in a straightforward way. In the model, the standard Dolev-Yao attacker model [6] is used, in which the attacker of a protocol is assumed to have full control of the network. It is also capable of replaying old messages, but can also construct new messages by decomposing previously sent messages into their parts and recombining those parts, possibly encrypted with known keys or other components. The model is based on the *strand space model* [7], but uses parameterised strands instead of constant strand definitions.

To model a security protocol, the roles of the principals that participate in a protocol are defined. A role definition serve as a schema for instantiations of principals. Two instances of a role are distinguished by the values of their parameters in a protocol analysis model. A *strand* is is a representation of an instantiated role. ACE takes as input a *semibundle*, which is a set of strands, and tries to find a protocol trace by instantiating the parameters of the strands in the semibundle. Such an instantiated set of strands is called a *bundle*. A semibundle can be modified in a number of ways to test different aspects of the protocol. For a complete analysis of a protocol using ACE, a vast number of different semibundle constructs must be processed. If a completed protocol execution is found, ACE outputs a protocol trace that describes the principals, their actions and the sequence of these actions.

IV. Dynamic Security Protocol Analysis with GRACE

Although possible, direct use of ACE requires good understanding of the model used in the engine and knowledge of the format in which the roles and semibundles are defined. Furthermore, the user would have to enumerate a large number of semibundle combinations, which is both tedious and error prone if done by hand. The **Gr**aphical **A**ttack **C**onduction **E**nvironment (GRACE) assists the user in performing a protocol analysis with ACE, so that even an engineer with limited insight in the analysis method can benefit from the system.

GRACE takes as input a protocol specification together with parameters for the analysis. The protocol specification is taken from from the SPEAR II specification environment, GYPSIE, and the analysis parameters are taken from the graphical user interface of GRACE. It returns either a trace of an abnormal protocol trace (an attack on the protocol) or no result at all (no attack found). This functional view of the module can be decomposed into subfunctions as in Figure 2.

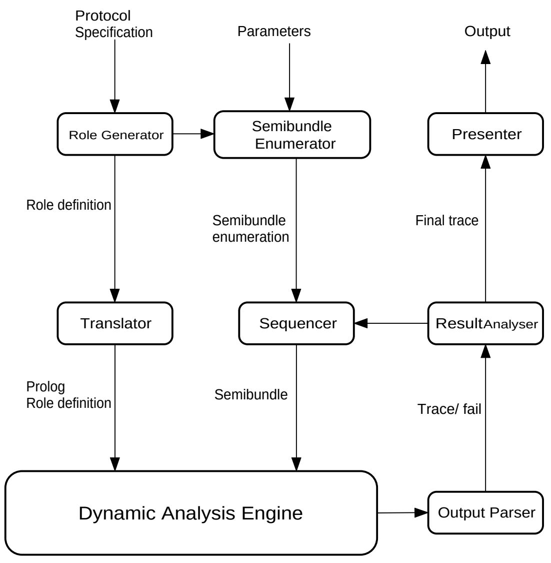

Fig. 2. Data flow in the dynamic analysis module

The protocol specification is used for the generation of the role definitions of the participants in the protocol. Components of the role definitions are also used in the semibundles that the analysis engine takes as input.

The *semibundle enumerator* provides a series of different semibundles for the analysis engine to attempt to solve. The most general case for the enumerator is to generate all possible permutations of principal instances in a protocol run. Without an upper bound on the number of instances, the enumeration would be infinite. Therefore the semibundle enumerator also takes as input a number of parameters that limit the size of its output.

The enumeration is split into individual semibundles that the analysis engine tests one at a time. This is done by the *sequencer*. Each role definition contains a message sequence in a SPEAR II format. In order for the analysis engine to understand the message sequences, they must be translated into a suitable format that the engine can understand. This task is carried out by the *role definition translator*.

At this stage, ACE has all the data it needs to perform a search on a semibundle. The output is either a trace of a protocol run, or a termination message. The *output parser* gathers the necessary information from the output and passes this information on to the *result analyser*. This function decides whether the trace that the engine returned is an abnormal protocol run. If the trace is normal, it tells the sequencer to continue the input to the analysis engine. If the trace represents an attack, the result analyser passes the result on to the presenter. If no trace was found, it reports this too. Finally, the *presenter* is a function that converts the trace into a format that SPEAR II can use.

The current implementation of GRACE is capable of producing input to ACE that detects secrecy violations for symmetric key protocol specifications, including those generated by type flaw attacks. However, only a subset of authentication failures can be detected. In order to detect the remaining attacks, a number of rules must be added to the semibundle enumeration procedure, to ensure that a complete search is performed.

V. The Protocol Code Generator

The **P**rotocol **C**ode **G**enerator (PCG) parses an abstract GYPSIE specification and generates a secure implementation of this specification. Java was chosen as the target language for a PCG implementation due to the language's excellent security architecture and resistance to common security attacks caused by buffer overflows as well as it's suitability to heterogenous networks. All message formatting in PCG is specified using ASN.1 [8]. Using an accepted standard such as this makes it possible for SPEAR II implementations to communicate with other non-SPEAR II implementations. Parameters and settings for code generation are specified in the graphical GENIE environment before PCG begins generation. PCG parses both the GYPSIE and GENIE data structures are also parsed by PCG in order to:

- generate code for any additional message processing actions specified
- extract message formats specified in ASN.1 and generate ASN.1 Java files.
- which cryptographic algorithms and library to use and
- what communication settings to use such as port number and transport protocol.

PCG does not build an intermediate parse tree from the protocol specification as other security protocol code generators such as *SPEAR I* [1], *AGVI* [9] and *Cryptographic Code Generation From CAPSL* [10] projects do. This is a benefit of the multi-dimensional approach followed in SPEAR II since the GYPSIE environment takes care of any semantic checking and symbol references so that all is required is the direct translation from the GYPSIE data structures to Java source code.

A. Generated Code Structure

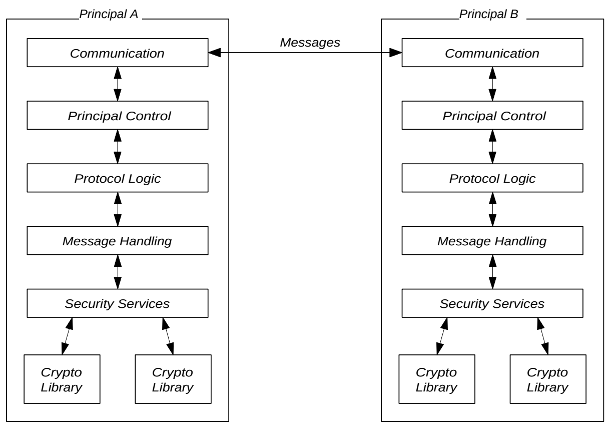

Fig. 3. The design and structure of protocol implementations.

Code generated by PCG follows the structure shown in Figure 3. The Communications layer handles all network connections for a principal and is part of the Java runtime library developed for SPEAR II. The Principal Control layer controls the actual principal application and the spawning of new threads. The Protocol and Message Handling layers are the actual protocol implementation are independent of the implementation which exists at the Principal Control and Communications layers. The *SecurityServices* layer is also part of the Java runtime library and acts as a common interface to the Cryptix [1] and Crypto-J [2] cryptographic libraries which are supported for generated implementations.

B. The PCG Generation Method

PCG contains the following public methods to initiate generation: `genPrincipals()`, `genProtocol()`, `genMsgHandlers()` and `genASN1Specification()`. The former two methods generate classes for the *Principal Control* and *Protocol Logic* layers in Figure 3 while the latter two are concerned with generating classes for the *Message Handling* layer. Code is generated by these methods using code templates distributed with SPEAR II which contain tokens that are replaced by generated code. An excerpt from a template file is shown below:

[1]The Cryptix library is available from http://www.cryptix.org
[2]Crypto-J is an RSA product, more information regarding it can be obtained from http://www.rsa.com

```
/* Prepare a message for sending */
public byte[] pack(int message)
{
  byte[] returnMsg = null;

  /* Pack routines for all messages sent */
  <\$MSGPACKROUTINES> ← A code template token

  return returnMsg;
}
```

The excerpt above is a simplified example from the *Message Handler* template. A token such as the one shown in the excerpt will be parsed and replaced by code generated to handle message processing for each message in the protocol. This approach is used for the generation of all the classes of a protocol implementation. The only exception to this being the ASN.1 Java classes for each message and message component.

The ASN.1 Java classes are generated by an external ASN.1 compiler, which takes a complete ASN.1 specification of all the messages and message components in a protocol as input. The ASN.1 compiler used for the current SPEAR II implementation is the ASN1C compiler [3]. PCG's job is to generate a complete ASN.1 specification from all the separate ASN.1 definitions specified for each message component described in Section VI. This is done without the use of any templates.

Code that PCG generates for performance analysis in SPANA (see Section VII) follows the same generation process as other code and the same classes are generated. The only difference is that PCG queries SPANA to determine what performance metrics to generate code for.

VI. The Graphical Code Generation Environment

Code cannot be generated in SPEAR II by relying on the GYPSIE protocol specification alone, since certain actions, such as checking the freshness of a nonce, are not explicitly specified in the GYPSIE specification. The **GEN**eration **E**nvironment (GENIE), therefore provides a method for the protocol engineer to control the implementation process, and be able to focus on the semantics of an implementation rather than the syntactical element in order to reduce the chance of programming errors and having to modify the generated code manually afterwards.

The GENIE environment comprises four interacting components, each of which is concerned with the specification of different areas of the implementation. Cryptographic settings, such as the cryptographic library used, are specified in the *Cryptographic Settings* component while parameters such as transport protocols and port numbers are specified in the *Communication Settings* component. The format of each message component and essentially, each message, is specified in the properties box of each message component. The *Message Processing Actions* component of the GENIE environment is concerned with the specification of both pre-processing and post-processing message actions such as checking the freshness of a nonce.

[3]ASN1C Compiler is a product of Objective Systems and more information can be found at http://www.obj-sys.com

The GENIE environment runs in tandem with the GYPSIE protocol specification environment and PCG. Sub-protocols, principals, messages and message components specified in GYPSIE are imported and used for constructing message processing actions and specifying cryptographic and communications settings. Parameters and settings are then parsed by PCG for source code generation. Results and progress indicators from the code generation process are retrieved from PCG so that they can be displayed appropriately.

GENIE allows a protocol engineer to specify cryptographic libraries and algorithms for each cryptographic operation in a SPEAR II protocol specification. Transport protocols and port numbers can also be specified for each principal. GENIE allows a protocol designer to specify both multi-threaded and single-threaded principals.

ASN.1 definitions are specified by the designer in GENIE for each message in the protocol specification. Message processing actions such as checking the freshness of a nonce are specified using an easy to use structured tree. Any message processing actions required such as checking the freshness of a nonce, are generated by default from the information produced for a GNY analysis in GYNGER, that is if such an analysis has been conducted. GENIE interacts closely with existing SPEAR II modules such as GYPSIE and the Visual GNY environment in order to carry the benefits gained from using GYPSIE and conducting a GNY logic analysis through to a protocol implementation.

VII. The Performance Analyser

The source code generated from a GYPSIE protocol specification can be augmented to provide performance measures for several aspects of a protocol implementation. The **S**PEAR II **P**erformance **ANA**lyser (SPANA) is the component of the SPEAR II Framework which interacts with the code generation module to produce executable source code for a specification and then gathers performance information from controlled source code executions and displays it to the protocol engineer.

A. Performance Measures Supported by SPANA

Each of the metrics belongs to a category most fitting to the aspect of a protocol that it measures. Under the *Protocol* category, a protocol engineer can choose to measure *Timings for the entire protocol execution* which will measure the total time taken for the complete execution of each protocol run and provide a mean time for all the protocol executions.

The metrics in the *Messages* category measure the following aspects of a protocol:

1) Timings for each principal's messages: This metric measures the time taken between the start of processing of one message to the start of processing of the next message for each message of the principal. The mean time for each message is returned as well as the mean time for all the principal's messages.

2) Timings for packing messages for sending: This metric measures the time taken to process a message for sending which includes cryptographic operations and ASN.1 encodings. The mean time for each message processing procedure in the protocol is returned as well as the mean time for all pack operations.

3) Timings for unpacking received messages: This measures the time taken to process a message that has been received which includes any ASN.1 decoding and decryption which must be performed. The mean time for each message unpacking procedure in the protocol is returned as well as the mean time for all unpack operations.

The metrics in the *Cryptographic* metrics category operate as follows:

1) Timings for all cryptographic operations: This provides mean and total times for all cryptographic operations in the entire protocol execution. This is ultimately a sum of all the timings for the cryptographic operations of each message and so provides an overall method of comparing cryptographic libraries. However it is not an entirely accurate comparison of cryptographic libraries since one library may be faster for an algorithm used in the first message while another is faster for an algorithm used in the second message and this measure provides an overall picture which ignores such subtleties.

2) Timings for cryptographic operations of principals: This measure provides mean and total times for all the cryptographic operations of each principal which is useful if the protocol engineer wishes to compare the time for two principals using different cryptographic libraries.

3) Timings for cryptographic operations of each message: This measure also provides both mean and total times for the cryptographic operations of each message. This measure is particularly useful since it can be used to compare the effect of using different cryptographic algorithms in a message and it provides a more accurate comparison of different cryptographic libraries since their times for each message's operations can be compared.

The most important contribution these metrics make to the performance analysis dimension is that they provide a basis from which to compare performance and hence act as comparison measures rather than precise measures of individual performance.

B. The SPANA Performance Analysis Method

A typical performance analysis session using SPANA starts with the specification of which metrics described above to measure. This is all done in a graphical user interface embedded in the SPEAR II tool. It is required that all code generation parameters are specified in GENIE first before a performance analysis can be conducted. Once the metrics the designer wishes to use have been specified the performance analysis can begin. SPANA will then invoke PCG to generate executable Java source code containing methods to write out measurements for the specified metrics which SPANA reads in and displays.

SPANA then invokes the SUN Java compiler to compile the PCG generated code and then begins executing the protocol. The number of protocol executions specified in the SPANA graphical interface prior to the start of analysis and determines the number of protocol runs that will be started and used to gather the performance measurements. Once all protocol executions are complete the mean values for the specified performance measurements are displayed in a tree-view of a results tab-sheet in the SPANA graphical interface. The results displayed in the tree-view follow a similar format to that shown below:

```
Category (e.g. Cryptographic Metric)
 ⇒Principal Name
    ⇒Message Name
       ⇒Measurement
```

However if the metric being measured is an overall metric like the *Protocol* metric then the *Principal Name* and *Message Name* nodes won't exist in the results tree and only the mean measurement will be shown.

The only results shown in the results tree-view are mean results for all protocol executions. If the protocol engineer would like more detailed measurements, another dialog can be triggered which shows the total figures for each measurement taken for all the protocol executions which are, essentially, the figures used to calculate the mean values in the results tree-view.

VIII. Conclusion

In this project, the SPEAR II Framework has been expanded and updated to realise a prototype for a complete attack analysis module as well as the addition of a code generation module and expansion of the performance analysis module. At the completion of this project the following modules had been integrated with the SPEAR II Framework:

- The GRACE attack analysis graphical interface (prototype).
- The ACE attack analysis engine (prototype).
- The GENIE code generation control and specification environment.
- The PCG code generator and translation engine.
- The SPANA performance analysis module.

A. Attack Analysis Dimension

The investigation into dynamic protocol analysis (attack analysis) methods and theory provided a solid basis from which to implement the prototype of the attack analysis dimension which includes the GRACE environment and the ACE attack analysis engine. The objective for the implementation of the attack analysis dimension from the start of the project was to show the usefulness of the available dynamic protocol analysis techniques in the context of a multi-dimensional protocol engineering tool. It is felt that this has effectively been shown through the capability of discovering protocol flaws that are undetectable in a static analysis such as GNY. Even though both the implementation and the chosen platform are far from optimal, the execution time for most protocol analyses is comparatively short, which is an argument in favour of the strand space algorithm that was used.

B. The Code Generation Dimension

The GENIE environment, through the use of intuitive and flexible graphical interfaces, assists the designer in abstractly specifying code generation settings, parameters and actions. The same graphical interfaces is used to illustrate the ease with which a protocol engineer can control the protocol implementation process with a degree of flexibility that allowed for many customised source code settings and actions to be specified without having to manually modify source code after the generation process.

The code generation process of the PCG Java generator was also described in detail and evidence was provided that PCG effectively provides a direct translation from the abstract data structures of GYPSIE and GENIE to a source code implementation. This means that the benefits of logical analysis and dynamic protocol analysis performed on a GYPSIE specification is effectively transferred through to a SPEAR II protocol implementation. All PCG generated code was developed by strictly following the guidelines outlined for secure Java development [11], thus increasing the trust in the security of SPEAR II implementations.

C. The Performance Analysis Dimension

The measurement of performance metrics in SPANA, such as the mean time to apply the cryptographic operations of a message, allows for useful comparisons to be made between the performance of different cryptographic libraries and algorithms. Other performance measures supported by SPANA assist the protocol engineer in identifying performance lags in the specified protocol as well as providing a comparison on the performance of different message structures. Another important contribution made by SPANA is the testing of how well protocol implementations handle concurrency as well as the number of connections capable of being supported before performance begins to lag. All of these measures and comparisons provided by SPANA give the protocol designer yet another perspective on the protocol engineering process which contributes to enhancing the efficacy and the efficiency of a protocol engineering and implementation session.

In combination with the existing modules, the described additions to the SPEAR II Framework form another step towards the implementation of a complete multi-dimensional security protocol engineering tool which will assist in producing more secure cryptographic protocols.

References

[1] J. Bekmann, P. D. Goede, and A. Hutchison, "SPEAR: Security Protocol Engineering and Analysis Resources," in *DIMACS Workshop on Design and Formal Verification of Security Protocols*. Rutgers University, September 1997.

[2] E. Saul and A. Hutchison, "SPEAR II: The Security Protocol Engineering and Analysis Resource," in *Second Annual South African Telecommunications, Networks and Applications Conference*, Durban, South Africa, September 1999, pp. 171 – 177.

[3] E. Saul, "Facilitating the modelling and automated analysis of cryptographic protocols," Master's thesis, DNA Research Group, Computer Science Department, University of Cape Town, 2001.

[4] L. Gong, R. Needham, and R. Yahalom, "Reasoning about Belief in Cryptographic Protocols," in *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*. Oakland, California: IEEE Computer Society Press, 1990, pp. 234 – 248.

[5] J. K. Millen and V. Shmatikov, "Constraint solving for bounded-process cryptographic protocol analysis," in *ACM Conference on Computer and Communications Security*, 2001, pp. 166–175.

[6] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198 – 208, 1983.

[7] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman, "Strand spaces: Why is a security protocol correct?" in *Proceedings of 1998 IEEE Symposium on Security and Privacy*, IEEE Comput. Soc, May 1998.

[8] *ITU-T Recommendation X.680 (1997): Abstract Syntax Notation One (ASN.1)*, International Telecommunication Union, Geneva, 1997.

[9] A. Perrig, D. Song, and D. Phan, "Agvi – automatic generation, verification, and implementation of security protocols," in *13th Conference on Computer Aided Verification (CAV)*, 2001.

[10] J. Millen, "CAPSL: Common authentication protocol specification language," The MITRE Corporation, Tech. Rep. MP 97B48, 1997.

[11] *Security Code Guidelines*, SUN Microsystems, http://java.sun.com/security/seccodeguide.html.