Analyzing Encryption Protocols Using Formal Verification Techniques

RICHARD A. KEMMERER, SENIOR MEMBER, IEEE

Abstract—**This paper presents an approach to analyzing encryption protocols using machine-aided formal verification techniques. The properties that the protocol should preserve are expressed as state invariants, and the theorems that must be proved to guarantee that the cryptographic facility satisfies the invariants are automatically generated by the verification system.**

A formal specification of an example system is presented, and several weaknesses that were revealed by attempting to verify and test the specification formally are discussed.

Introduction

MUCH work has been done in the area of analyzing encryption algorithms, such as DES [2]–[4], [7]. A vast amount of work has also been expended on formally verifying communication protocols [13], [14], [17], [23], [26]. In contrast, very little work has been devoted to the analysis and formal verification of encryption protocols, and the work that has been presented has not made use of the available formal verification systems [1], [10], [21]. In this paper an approach to analyzing encryption protocols using machine-aided formal verification techniques is presented.

When considering a secure network that uses encryption to achieve its security (as opposed to using only physical security), one must consider both encryption algorithms and encryption protocols. An *encryption algorithm*, such as DES or RSA, is used to convert clear text into cipher text or cipher text into clear text. That is, the unencrypted message (clear text) is enciphered using a particular encryption algorithm to produce the unreadable cipher text. Similarly, the same or a symmetric algorithm is used to recover the original clear text message from the encrypted cipher text. An *encryption protocol* is a set of rules or procedures for using the encryption algorithm to send and receive messages in a secure manner over a network. The approach presented in this paper does not attempt to prove anything about the strength of the encryption algorithms being used. On the contrary, it may assume that the obvious desirable properties, such as that no key will coincidentally decrypt text encrypted using a different key, hold for the encryption scheme being employed.

The idea of the approach to analyzing encryption protocols that is presented in this paper is to specify formally the components of the cryptographic facility and the associated cryptographic operations. The components are represented as state constants and variables, and the operations are represented as state transitions. The desirable properties that the protocol is to preserve are expressed as state invariants, and the theorems that must be proved to guarantee that the system satisfies the invariants are automatically generated by the verification system.

The following section reviews past work on the analysis of encryption protocols. This is followed by a brief overview of the formal specification language that is used. Next, a sample system is presented along with the desirable cryptographic properties for that system. A formal specification for the example system is then given, followed by a discussion of formally verifying and testing encryption protocol specifications. A weakness of the example specification that was discovered through testing the specification is presented, and a further weakness using semiweak DES keys is also discussed. Finally, a comparison to other work in this area is given, and some thoughts on the usefulness of the approach presented in this paper are discussed.

Previous Work

One of the earliest and best-known works devoted to the analysis of encryption protocols is that of Dolev and Yao [10], which concentrated on two-party public key protocols. In this work, the authors pointed out that although a protocol may be secure against passive attacks, such as an eavesdropper, it may be vulnerable to active attacks. In their work, they develop mathematical models for cascade and name-stamp protocols and propose algorithms to test if a protocol of either of these types is secure.

Book and Otto [1] later extended the work of Dolev and Yao to consider message authentication as well as security. In this work, algorithms for determining whether a two-party protocol is sender verifiable or receiver verifiable are presented.

Recently reported work by Moore [21] analyzes a number of previously known encryption protocol failures. After analyzing these failures, Moore then describes three distinct classifications for encryption protocol failures.

Manuscript received August 1, 1988; revised January 15, 1989. This research was supported in part by the System Development Group of Unisys Corporation and in part by the University of California under a MICRO grant.

The author is with the Department of Computer Science, University of California, Santa Barbara, CA 93106.

IEEE Log Number 8927170.

She also presents some guidelines for the development of sound protocols based on her analysis.

Formal specification and verification techniques have become an accepted technique for ensuring that a critical system satisfies its requirements. In fact, the National Computer Security Center, which certifies systems for use in classified or other sensitive environments, requires formal specification and verification of system designs for its highest rating [9]. There are, however, some in the security community that still view with suspicion the use of formal verification techniques to achieve reliable software. The most vocal of these are DeMillo, Lipton, and Perlis [8], who argue that "it is a social process that determines whether mathematicians feel confident about a theorem—and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail." They also state that "scientists should not confuse mathematical models with reality—and verification is nothing but a model of believability." Another often-cited reference to the shortcomings of formal verification techniques appeared in Thompson's 1983 Turing Award lecture [28], where he said "no amount of source-level verification or scrutiny will protect you from using untrusted code." Thompson is specifically referring to a flaw that he had put in the C compiler that would allow him to subvert the login command. Because the offending source code was removed after its use, there was no trace in the source. Thus, it would be necessary to verify formally the object code to find the problem. Although there is some validity to these arguments, formal specification and verification techniques should not be abandoned until there is a better method to replace them.

Formal specification and verification techniques have often been used in secure operating systems and communication protocol efforts. There are also a number of formal verification systems currently available [6], [12], [22], [24], [27]. These systems all use mathematical techniques to guarantee the correctness of the system being designed and implemented. To use these techniques, it is necessary to have a formal notation, which is usually an extension of first-order predicate calculus, and a proof theory.

The three encryption protocol analysis efforts reported above do not use formal verification techniques. There have, however, been several efforts reported in the literature that do use formal verification techniques to model secure networks [5], [16], [29], although none of these efforts models active attacks. That is, they do not consider the active intruder.

A notable exception is the Interrogator work of Millen *et al.* [20]. The Interrogator tool was built explicitly for analyzing encryption protocols. It uses a Prolog specification as a formal notation to express the encryption protocol. When using the Interrogator, the Prolog program exhaustively searches for penetrations. The Interrogator also includes a sophisticated display that dynamically illustrates the progress of the protocol being tested.

The recently reported work of Longley [18] uses an expert system and is similar to the Interrogator approach in that it is rule based and also performs an exhaustive search once the rules are all defined. It also considers possible penetrations and tries those.

The approach reported here differs from Millen's and Longley's work in that the goal is to use an existing formal verification approach and its associated tools to verify formally that an encryption protocol specification satisfies its stated security requirements or, alternatively, to discover weaknesses in the specification. The formal verification system and language that are used are presented in the next section.

The Formal Specification Language

The formal verification system discussed in this paper uses the state machine approach to formal specification. When using the state machine approach, a system is viewed as being in various states. One state is differentiated from another by the values of state variables, and the values of these variables can be changed only via well-defined state transitions.

The formal specification language that is used is a variant of Ina Jo®, which is a nonprocedural assertion language that is an extension of first-order predicate calculus. The key elements of the Ina Jo language are types, constants, variables, definitions, initial conditions, criteria, and transforms. A criterion is a conjunction of assertions that specify the critical requirements for a good state (i.e., a secure state). A criterion is often referred to as a state invariant since it must hold for all states, including the initial state. An Ina Jo language transform is a state transition function; it specifies what the values of the state variables will be after the state transition relative to their values before the transition. The system being specified can change state only as described by one of the state transforms. A complete description of the Ina Jo language can be found in the Ina Jo Reference Manual [24].

Before giving a formal specification of the example system, a brief discussion of some of the notation is necessary. The following symbols are used for logical operations:

& logical AND,
→ logical implication.

In addition, there is a conditional form,

(if A then B else C),

where A is a predicate and B and C are well-formed terms. The notation for set operations is:

$\in$	is a member of,
$\cup$	set union,
$\{a, b, \ldots, c\}$	set consisting of elements a, b, . . . , and c,
{set description}	set described by set description.

®Ina Jo is a trademark of the System Development Group of Unisys.

The language also contains the following quantifier notation:

- ∀ for all,
- ∃ there exists.

Two other special Ina Jo symbols that may be used are

- N'' to indicate the new value of a variable (e.g., $N''v1$ is new value of variable $v1$),
- T'' which defines a subtype of a given type T.

An Example System

The system being used as a pedagogical example in this paper is a single-domain communication system using dynamically generated primary keys and two secret master keys, as described in [19]. The architecture of the system is presented in Fig. 1.

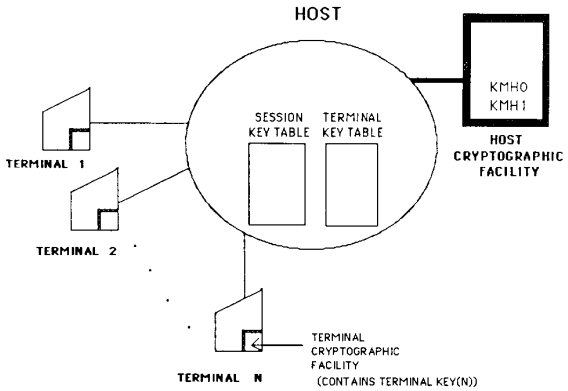

Fig. 1. System architecture.

The host manages the communication keys for the system, and the terminals communicate directly with the host system. Whenever a terminal wants to start a new session with the host, the host generates a new session key. If this key were sent to the terminal in the clear, a penetrator tapping the line could intercept the key and decipher all of the messages for that session. To prevent this, each terminal has a permanent terminal key that is used by the host to distribute the new session key to a terminal when a new session is initiated. That is, a terminal's session key is the primary communication key for that terminal. It is dynamically generated by the host for each session. The static terminal key is the terminal's secondary communication key. It is used by the host to encrypt new session keys for transmission to the terminal.

Both the terminal keys and the current session keys are stored at the host. However, because a penetrator can be an authorized user, it is unsafe to store the terminal and session keys in the clear at the host. Thus, they are stored in encrypted form. The two host master keys are used for encrypting these keys. The following paragraphs give the details of the key table structures and the cryptographic facility.

There are two data structures of interest in the host: the terminal key table and the session key table. The *terminal key table* is static. Each entry in this table contains the unique terminal key for the corresponding terminal encrypted using a secret master key KMH1. The table looks as follows:

E_{KMH1}(Terminal Key(1))
E_{KMH1}(Terminal Key(2))
:
E_{KMH1}(Terminal Key(i))
:
E_{KMH1}(Terminal Key(n))

In this paper, $E_{\text{key-name}}$ (text) is used to denote text encrypted using the key key-name. Similarly, $D_{\text{key-name}}$ (text) is used to denote text decrypted using the key keyname. Since terminal keys never change, this table is constant for the lifetime of the system.

Unlike the terminal key table, the *session key table* is a dynamic structure. This table is updated each time a new terminal session is started; there is one current session key per terminal. Each entry in the table contains the current session key for the corresponding terminal encrypted using a second secret master key KMH0. The session key table looks as follows:

E_{KMH0}(Session Key(1))
E_{KMH0}(Session Key(2))
:
E_{KMH0}(Session Key(i))
:
E_{KMH0}(Session Key(n))

No terminal key or session key and neither master key is in the clear in the host. To store the two masters keys, a *cryptographic facility* is connected to the host. This facility may be accessed only through the limited cryptographic operations that are provided. In addition, the facility is assumed to be housed in a tamper-sensing container, such as the one described by Simmons [25], so that the vital information it contains is physically protected. The operations provided by the cryptographic facility are encipher data (ECPH), decipher data (DCPH), and reencipher from the master key (RFMK). The interaction between the host and its cryptographic facility is shown in Fig. 2.

The *encipher* operation is used when the host wants to send an encrypted message to a terminal. The host provides the clear text message (msg) along with the encrypted form of the appropriate terminal's current session key, E_{KMH0} (Session Key(i)), to the cryptographic facility and is returned the message encrypted using the terminal's session key. Fig. 3 illustrates this process.

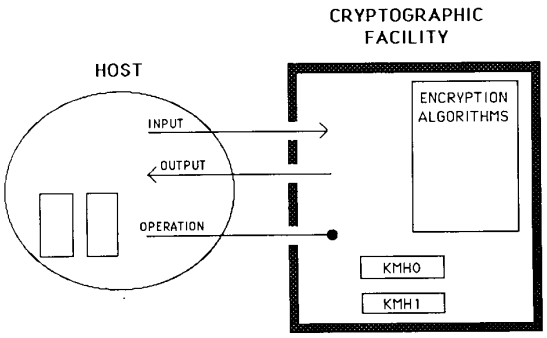

Fig. 2. Host and cryptographic facility.

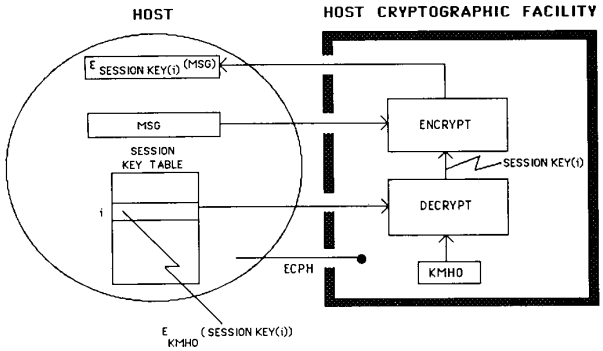

Fig. 3. The encipher operation (ECPH).

When the host receives an encrypted message from terminal i, it uses the *decipher* operation provided by the cryptographic facility in a similar manner to get the message in the clear. That is, the decipher operation must first decrypt the key presented and then use the result to decipher the text presented. Fig. 4 illustrates the decipher process.

Each time a terminal initiates a session with the host a new session key is needed. Therefore, the host needs access to a *generate session key* operation. It is not necessary, however, to have session key generation be an operation provided by the cryptographic facility. The seemingly contrary requirements of having the host generate the session key and having no key in the clear outside the cryptographic facility are resolved by having the host generate an encrypted version of the session key. The entry in the host's session key table that corresponds to the terminal requesting a session is then replaced by the encrypted key. Meyer and Matyas describe a method of accomplishing this by using a pseudorandom number generator to generate a value that is interpreted as the new session key encrypted using the secret master key KMH0 [19].

Since the requesting terminal is also sent a copy of the session key, it is necessary to have an operation to translate the new session key enciphered using the KMH0 master key to a form enciphered using the requesting terminal's terminal key. The *reencipher from master key* operation provides this service. It takes two keys as input,

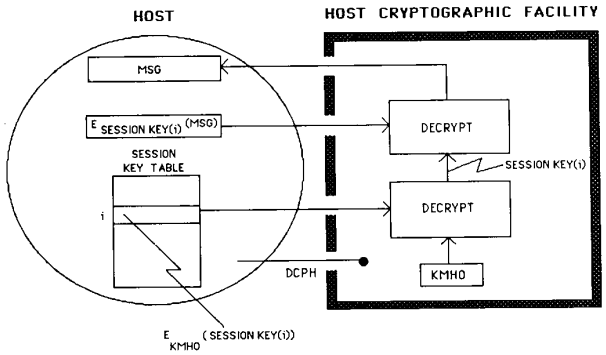

Fig. 4. The decipher operation (DCPH).

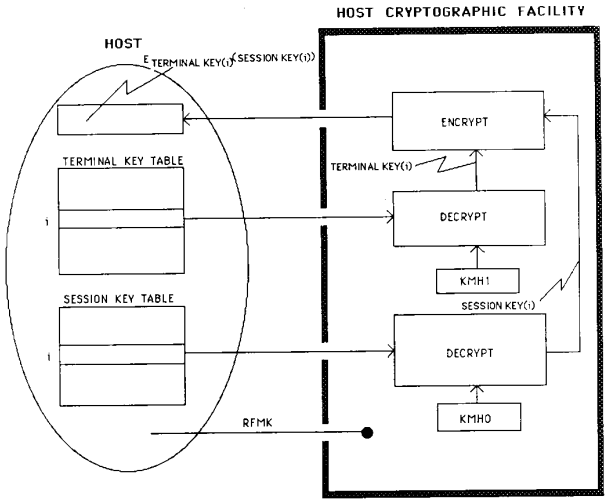

Fig. 5. Reencipher from master key (RFMK).

decrypts one using KMH1, decrypts the other using KMH0, and then uses the result of the first decryption to encipher the result of the second decryption. Fig. 5 illustrates this process.

In addition to the host's cryptographic facility, each terminal is assumed to have a cryptographic facility that contains its permanent terminal key and that provides operations for encrypting and decrypting messages.

Operations for setting the secret master keys in the host's cryptographic facility or the secret terminal keys in the terminal cryptographic facilities have intentionally been avoided in this paper. It is assumed that these secret keys are distributed by courier or some other trusted means.

An assumption of this system is that the intruder can eavesdrop on the communication lines. That is, the intruder can obtain any information communicated between the host and the terminals. In addition, the intruder can masquerade as an authorized user, and he/she can invoke any of the operations of the host's cryptographic facility.

An obvious desirable property of this system that one may wish to verify is that no clear key exists outside the cryptographic facilities of the host and terminals.

Formal Specification of the Example System

The complete Ina Jo specification for the example system is presented in the Appendix. In this section, the important aspects of the specification are discussed.

In the example system, each terminal has a constant terminal key. This is represented in the model by the parameterized Ina Jo constant

Terminal_Key(Terminal_Num): Key.

Similarly, each terminal's session key, which is dynamic, is represented by the parameterized Ina Jo variable

Session_Key(Terminal_Num): Key.

As the terms imply, an Ina Jo *constant* is unchanged from state to state, and an Ina Jo *variable* may change from state to state. It is the values of the state variables that differentiate one state from another.

The other state variables in the specification are the sets Keys_Used and Intruder_Info, which are both of type information. Keys_Used indicates all of the keys that have ever been used by the system. The Intruder_Info variable tabulates all of the information to which the intruder has access. This includes the contents of the encrypted key tables as well as any information communicated between the host and the terminals.

The four cryptographic operations for the example system are represented by the Ina Jo *transforms* ECPH, DCPH, RFMK, and Generate_Session_Key. The first three correspond to the operations provided by the host's cryptographic facility, and the last is provided by the host itself. Since only traffic that is a key or an encrypted form of a key is of interest, there is no need to model any of the send or receive text operations. It is also assumed that the intruder cannot correctly guess random text that corresponds to some encrypted key.[1] Therefore, the ECPH, DCPH, and RFMK operations change state only when they are invoked with information that the intruder has available. Thus, the state transitions that correspond to these operations are written using a conditional form where there is no state change if the information provided is not available to the intruder (i.e., not an element of the set Intruder_Info).

The constants Encrypt and Decrypt represent the encryption and decryption algorithms, respectively. They both take a key and text pair and return text. Properties of the encryption and decryption algorithms are represented in the specification as axioms. An Ina Jo *axiom* is an expression of a property that is assumed. Thus, these qualities are assumed about the algorithms. For instance, one could express the fact that the encryption and decryption functions were commutative by using the axiom

AXIOM ∀ t:Text, $k1$,$k2$:Key (Encrypt($k1$,Decrypt ($k2$,t)) = Decrypt($k2$,Encrypt($k1$,t))).

Similarly, the following axioms express the properties that no key is an identity function for any text and that no key will correctly decrypt text encrypted using another key.

AXIOM ∀ t:Text, $k1$:Key (Encrypt($k1$,t) ≠ t)

and

AXIOM ∀ t:Text, $k1$,$k2$:Key ($k1$ ≠ $k2$ → Decrypt($k2$,Encrypt($k1$,t)) ≠ t).

Note that all of these assumptions are not expressed in the example specification.

The fact that the intruder receives all of the information that is communicated between the host and the terminals is expressed in the Generate_Session_Key transform, where the intruder's information is enhanced with the new session key encrypted using the terminal key of the requesting terminal. Also, since the intruder can masquerade as an authorized user, whenever one of the cryptographic operations is invoked, the intruder's information is updated with the new information that is produced.

The critical requirements that the system is to satisfy in all states are expressed in the *criterion* clause of the formal specification. For the example system, the criterion states that any key that the intruder has (i.e., any key contained in the set Intruder_Info) must not be a key that was used by the system (i.e., a key in the set Keys_Used). Note that this includes keys used in the past as well as those presently being used. The criterion is expressed as follows:

CRITERION ∀ k:Key (k ∈ Intruder_Info → k ∉ Keys_Used).

The *initial* clause describes the requirements that must be satisfied when the system is initialized. For the example system, the initial value of the Keys_Used variable is the set of keys currently being used (i.e., all terminal keys and session keys as well as both master keys). The initial value of the Intruder_Info variable is the appropriately encrypted versions of the terminal and session keys. Because the keys are not required to have any particular value, their values are not specified in the initial clause. However, since the desirable property is for the intruder never to have any keys in the clear, the last conjunct of the initial clause states that none of the encrypted values of the keys can be coincidentally equal to a key being used. That is,

∀ $k1$,$k2$:Key ($k1$ ∈ Intruder_Info & $k2$ ∈ Keys_Used → $k1$ ≠ $k2$).

The need for this additional requirement is discussed in the next section.

To verify that the system specified satisfies the invariant requirements, as expressed in the criteria, two types of theorems are generated. The first states that the initial state satisfies the invariant, and the second, which is generated for each transform, states that if the state where the transform is fired satisfies the invariant, then the resultant state will also satisfy the invariant. Thus, by induction, all reachable states will satisfy the invariant.

[1] This is a reasonable assumption because the encryption algorithms are assumed to be as strong as advertised.

If one can verify the theorems generated, then any system that is consistent with the specification will preserve the invariant. The reader should note that, for a system to be consistent with the specification, its encryption algorithms must satisfy the axioms stated about encrypt and decrypt.

Formally Verifying the Specification

After the formal specification is completed, one can verify the theorems that are generated to check if the critical requirements (Ina Jo criteria) are satisfied. If the theorems are verified and the encryption algorithms satisfy the assumed axioms, then the system will satisfy its critical requirements.

Because the axioms represent the properties that the encryption algorithms are to satisfy, one can verify the system assuming the use of a different encryption scheme by replacing the current axioms with axioms that express the properties of the new encryption scheme.

An advantage of expressing the system using formal notation and attempting to prove properties about the specification is that, if the generated theorems cannot be proved, the failed proofs often point to weaknesses in the system or to an incompleteness in the specification. That is, they often indicate the additional assumptions required about the encryption algorithm (i.e., missing axioms), weaknesses in the protocols, or missing constraints in the specification. For example, the original specification for the example system did not include the third conjunct that is now in the initial clause. However, without this conjunct, the initial clause was not strong enough to imply the invariant. After analyzing the failed proof, the possibility of an encrypted version of a key being coincidentally identical to another key was apparent. By adding the third conjunct to the initial clause, the problem was avoided. This was a reasonable change to make to the specification since the occurrence of coincidental values is easy to check when the system is initialized.

Being aware of the coincidental key value problem in the initial clause resulted in a strengthening of the specification for the Generate Session Key operation. That is, the requirements

$$\text{Encrypt}(\text{KMH0},k) \notin \text{Keys\_Used}$$

and

$$\text{Encrypt}(\text{Terminal\_Key}(\text{Ter}),k) \notin \text{Keys\_Used}$$

were added to the formal specification to prevent the encrypted value chosen as a new session key from being coincidentally equal to the value of a key that had been used in the past. This requirement is likely to be harder to realize in an actual system since it requires recording information about all keys that have been used over the entire lifetime of the system.

Testing the Formal Specification

There is a specification execution tool for the Ina Jo language called Inatest [11]. This tool allows Ina Jo specifications to be analyzed by symbolically executing the formal specifications. With the Inatest tool, it is possible to introduce assumptions about the system interactively, execute sequences of transforms, and check the results of these executions. This provides the user with a rapid prototype for testing properties of the cryptographic facilities [15].

Using the Inatest tool revealed the following weakness in the example formal specification. If the secret master keys KMH0 and KMH1 are equal, then the intruder can obtain a session key in the clear. This flaw is demonstrated by first simulating an innocent user invoking the Generate_Session_Key transform, which generates a new session key k. This key is communicated to the requesting terminal (encrypted using the terminal key of the requesting terminal) at the start of the current session; therefore, the encrypted key becomes part of the eavesdropping intruder's information. The DCPH transform is then invoked by the intruder using the encrypted terminal key from the host's terminal key table and the intercepted session key encrypted using the requesting terminal's terminal key as text. The result is that the session key of the innocent user is now in the clear, and any messages encrypted using this key can now be deciphered. Fig. 6 illustrates the result of executing the DCPH transform on the two encrypted keys. Note that the first decrypt yields the appropriate terminal key only because KMH0 = KMH1.

When using the Inatest tool to test a formal Ina Jo specification, the user defines a start state, a sequence of transforms (with the appropriate actual parameters) to be executed, and a desired resultant state. To test this weakness, the default *start state*, which is the initial state, was used, and it was also assumed that KMH0 = KMH1.

$$\begin{aligned}
&\text{Keys_Used} = \text{Terminal_Keys} \cup \text{Session_Keys} \cup \{\text{KMH0},\text{KMH1}\} \\
&\& \ \text{Intruder_Info} = \\
&\quad \{ks{:}\text{Key} \ (\exists\, t{:}\text{Terminal_Num} \\
&\quad (ks = \text{Encrypt}(\text{KMH0},\text{Session_Key}(t))))\} \\
&\quad \cup \{kt{:}\text{Key} \ (\exists\, t{:}\text{Terminal_Num} \\
&\quad (kt = \text{Encrypt}(\text{KMH1},\text{Terminal_Key}(t))))\} \\
&\& \ \forall\, k1,k2{:}\text{Key} \ (k1 \in \text{Intruder_Info} \ \& \ k2 \in \text{Keys_Used} \\
&\qquad \rightarrow k1 \neq k2) \\
&\& \ \text{KMH0} = \text{KMH1}.
\end{aligned}$$

The *sequence of transforms* to be executed consists of the Generate_Session_Key transform followed by the DCPH transform. The parameters of the DCPH transform are the encrypted terminal key for terminal t and the current session key for terminal t encrypted using the terminal key for terminal t. Both keys are known to be part of the intruder's information. The first is from the terminal key table, and the second was sent to terminal t when the current session was started. Letting k represent the key that results from executing the Generate_Session_Key transform on behalf of terminal t, the sequence is,

$$\text{Generate\_Session\_Key}(t)$$

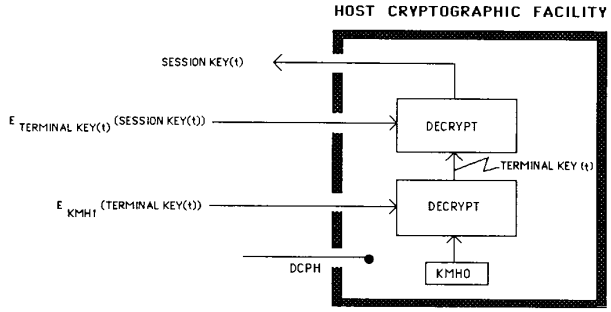

Fig. 6. Protocol flaw using DCPH.

followed by

DCPH(Encrypt(KMH1,Terminal_Key(t)),
Encrypt(Terminal_Key(t),k)).

The desired *resultant state* requires that the key for terminal t that was generated by the Generate_Session_Key transform be part of the intruder's information. This requirement is expressed as

$k \in$ Intruder_Info.

This is a clear violation of the security requirement since k is one of the keys used by the system.

By expanding Generate_Session_Key, one gets

$\exists$ k:Key $\forall$ $t1$:Terminal_Num (
Encrypt(KMH0,k) $\notin$ Keys_Used
& Encrypt(Terminal_Key(t),k) $\notin$ Keys_Used
& $k \notin$ Keys_Used
& N"Session_Key(t) =
(if $t1 = t$
then k
else Session_Key($t1$))
& N"Keys_Used = Keys_Used $\cup$ {k}
& N"Intruder_Info = Intruder_Info $\cup$
{Encrypt(KMH0,k),Encrypt(Terminal_Key
(t),k))}.

The existential is instantiated to k, and the resulting information is combined with the start state information.

Next, by expanding the DCPH transform, one gets

N"Intruder_Info =
(if Encrypt(Terminal_Key(t),k) $\in$ Intruder_Info
& Encrypt(KMH1,Terminal_Key(t)) $\in$ Intruder_Info
then Intruder_Info $\cup$
{Decrypt(Decrypt(KMH0,Encrypt(KMH1,
Terminal_Key(t))),Encrypt(Terminal_
Key(t),k))}
else Intruder_Info).

Since both keys are part of the intruder's information, this can be reduced to

N"Intruder_Info = Intruder_Info $\cup$
{Decrypt(Decrypt(KMH0,Encrypt(KMH1,
Terminal_Key(t))),Encrypt(Terminal_
Key(t),k))}.

However, the start state specifies that KMH0 = KMH1; therefore, KMH1 can be substituted for KMH0 in the innermost Decrypt, yielding

N"Intruder_Info = Intruder_Info $\cup$
{Decrypt(Decrypt(KMH1,Encrypt(KMH1,
Terminal_Key(t))),Encrypt(Terminal_
Key(t),k))}.

Then, by applying the first axiom, the expression reduces to

N"Intruder_Info = Intruder_Info $\cup$
{Decrypt(Terminal_Key(t),Encrypt(Terminal_
Key(t),k))}.

Applying the first axiom again yields

N"Intruder_Info = Intruder_Info $\cup$ {k}.

The desired result follows directly.

This is a well-known weakness of using a single master key that is presented in [19]. To strengthen the specification to avoid this particular problem, one needs to add the axiom

AXIOM KMH0 $\neq$ KMH1.

Semiweak Key Modification to the Example System

As was mentioned earlier, the work reported in this paper assumes that the encryption algorithms perform as promised. That is, nothing is proved about the strength of the encryption algorithms. One can use the Inatest system, however, to analyze the effect of weaknesses in the encryption algorithm. For example, one could analyze the effect of using semiweak key pairs with the DES algorithm and the example system. Semiweak keys as presented in [7] are two keys $k1$ and $k2$, such that, for any clear text t, Encrypt($k2$,Encrypt(k,1,t)) = t. This can be added to the example specification by using the following axiom:

AXIOM $\exists$ $k1$,$k2$:Key $\forall$ t:Text
(Encrypt($k2$,Encrypt($k1$,t)) = t).

Given this axiom, it is trivial to demonstrate how the system can be compromised.

To demonstrate an example compromise, the Inatest tool was used. When using this tool, one can test hypothetical situations like this without changing the original specification. This is accomplished by using the add command to add a predicate to the known information. For example, suppose the secret master key KMH1 and some session key formed a semiweak key pair. This information could be added to the knowledge base of Inatest by using the add command and the following predicate:

$\exists$ i:Terminal_Num $\forall$ t:Text
(Encrypt(Session_Key(i),Encrypt(KHM1,t)) = t).

Now, the intruder posing as terminal i can invoke the encipher operation using any of the stored encrypted termi-

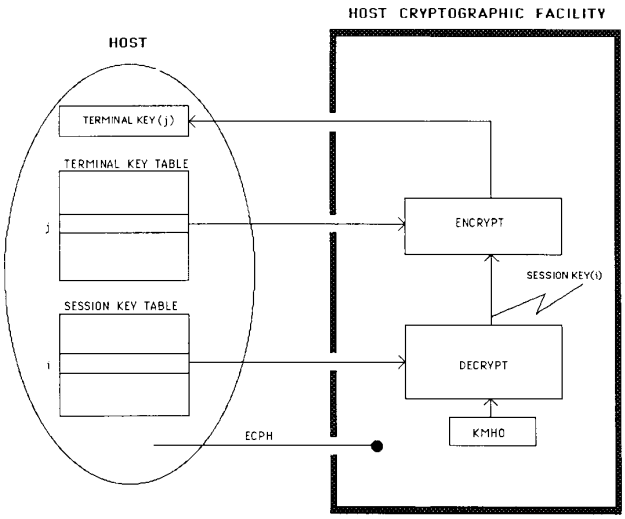

Fig. 7. Semiweak key DES key flaw using ECPH.

nal keys as the message and get the corresponding terminal key in the clear. Fig. 7 illustrates this flaw. By repeating this for all of the encrypted terminal keys in the terminal key table, the intruder can obtain all of the terminal keys in the clear.

Comparison to Other Work

As was mentioned in the previous work section, very little research has been devoted to the analysis and formal verification of encryption protocols. In particular, formal verification techniques have not been used in most of the analysis efforts that have been reported. Like the work of Dolev and Yao and Book and Otto, the approach reported in this paper considers the active attacks as well as the passive attacks. That is, the model includes the intruder as part of the formal specification.

The approach outlined in this paper is closest to the work of Millen *et al.* [20]. However, the work reported in this paper differs from Millen's work in that the goal of the work being reported is to use existing formal verification tools to verify formally that an encryption protocol specification satisfies its security requirements (as expressed in the Ina Jo criteria). This is accomplished by using the existing Formal Development Methodology (FDM) tool suite and treating the encryption protocol specification like any Ina Jo formal specification. The two efforts are similar in that they both use a formal notation to express the protocol.

The use of the Inatest tool for testing particular scenarios is also similar to the use of the Interrogator tool. However, there are at the same time major differences between using Inatest to test a protocol and using the Interrogator. When using the Interrogator, the Prolog program exhaustively searches for penetrations. Inatest, in contrast, does not search through a large number of scenarios to detect a vulnerability. It is the task of the human analyzer to come up with a possible scenario, which is then checked using the Inatest tool to execute the formal specification. Inatest does not direct the analyst to determine what tests to try; it merely aids the analyst by keeping track of state information and performing reductions when possible.

Like the Interrogator, Longley's rule-based expert system also performs an exhaustive search using the rules that define the protocol. However, it can also be used in a manner similar to that of the Inatest tool in that specific penetrations can be tried just like a test case using Inatest. Longley says that his system provides "A working model upon which experiments can be conducted." This is exactly the goal of the Inatest system.

In comparing the work reported in this paper to the work of Millen *et al.* and Longley, three approaches to analyzing an encryption protocol can be identified. They are

1) formally verify that the protocol specification satisfies the desired critical requirements,
2) exhaustively test the protocol, hoping to uncover a flaw, and
3) test a particular scenario to try to uncover or substantiate a flaw.

The work reported in this paper uses both 1) and 3), Millen and Longley use 2), and Longley, to some extent, also uses 3).

Clearly, if approach 1) could be achieved, it would be the most desirable approach. However, it is often impossible to prove formally the theorems generated from the protocol specification. An example of a problem that was encountered when trying to prove a particular protocol is the need for the formulation of the notion that attempting to decrypt with the wrong key gives no additional information. Rather, it adds another layer of encryption to the already-encrypted text. That is, if $k1 \neq k2$, then

$$\text{Decrypt}(k2,\text{Encrypt}(k1,t))$$

is equivalent to

$$\text{Encrypt}(k3,\text{Encrypt}(k1,t))$$

for some key $k3$. The concise expression of concepts like these is difficult, but without them the proofs are not possible. The earlier example involving encrypted keys that were coincidentally equal to clear keys also required the formulation of a concept before the protocol proofs could be attempted.

Approach 2), the exhaustive search approach, will reveal protocol flaws if they exist. However, as is the case with exhaustive testing, there is some concern about the use of this approach on more complex encryption protocols for which it may not be feasible to try all paths.

Approach 3) suffers from all of the standard problems of testing. In particular, if one does not test the appropriate scenario, no flaws will be revealed. It is our experience, however, that if one has difficulty in trying to prove one of the theorems generated using approach 1), then these failed proofs often lead to scenarios that can be tested using approach 3). Experience shows that these scenarios are likely to reveal flaws in the encryption pro-

tocol. A combination of all three approaches is likely to be most fruitful.

Finally, the work reported here is similar to that of Moore [21] in that it is anticipated that, by formally analyzing encryption protocols, guidelines can be established for the design of encryption protocols for secure networks.

Conclusions

This paper has proposed an approach to analyzing encryption protocols using formal specification and verification techniques and existing tools. With this approach, nothing is proved about the encryption algorithms. That is, encryption algorithms that satisfy the properties expressed in the axioms are assumed to be available.

A single-domain communication system with dynamically generated primary keys was formally specified using the Ina Jo specification language. Some problems discovered when attempting to prove the original specification were discussed, and a weakness in the formal specification that was revealed by using an interactive testing tool was presented. A second flaw using semiweak DES keys was also presented.

An advantage of this approach is that the properties of a cryptographic facility can be tested before the facility is built. First, the system is represented using a formal notation. The resulting formal specification is used to generate the necessary theorems that ensure that the system satisfies certain desired properties. If the generated theorems cannot be proved, then the failed proofs often point to weaknesses in the system or to an incompleteness in the specification. That is, they often indicate the additional assumptions required about the encryption algorithm, weaknesses in the protocols, or missing constraints in the specification.

Another advantage of this approach is that the cryptographic facility can be analyzed assuming different encryption algorithms by replacing the set of axioms that express the properties assumed about the encryption algorithms with a new set of axioms that express the properties of a different encryption algorithm. This was the approach used when analyzing the use of semiweak DES keys.

The first flaw that was presented was a previously known weakness of the protocol being analyzed, and the second would be obvious to anyone familiar with DES semiweak keys. The value of the proposed approach will be demonstrated when a flaw is discovered in a protocol that has previously been assumed to be secure.

Appendix
Formal Specification of the Example System

```
SPECIFICATION Crypto
LEVEL Top_Level

TYPE
   Text,
   Key subtype Text,
   Pos_Integer = T" i:Integer (i>0),
   Information = Set Of Text

CONSTANT
   Num_Terminals: Pos_Integer,
   KMH0, KMH1: Key,
   Encrypt(Key,Text): Text,
   Decrypt(Key,Text): Text

TYPE
   Terminal_Num =
     T" t:Pos_Integer (t < = Num_Terminals)

CONSTANT
   Terminal_Key(Terminal_Num): Key,
   Terminal_Keys: Information =
      {k:Key ( ∃ t:Terminal_Num
        (k = Terminal_Key(t)))}

AXIOM
   ∀ t:Text, k1,k2:Key
      (k1 = k2 → Decrypt(k1,Encrypt(k2,t)) = t )

AXIOM
   ∀ t:Text, k1,k2:Key
      (k1 = k2 → Encrypt(k1, Decrypt(k2,t)) = t )

VARIABLE
   Session_Key(Terminal_Num): Key,
   Keys_Used: Information,
   Intruder_Info: Information

DEFINE
   Session_Keys: Information = =
      {k:Key ( ∃ t:Terminal_Num
        (k = Session_ Key(t)))}

CRITERION
   ∀ k:Key (k ∈ Intruder_Info → k ∉ Keys_Used)

INITIAL
   Keys_Used = Terminal_Keys ∪ Session_Keys ∪
      {KMH0,KMH1}
 & Intruder_Info =
      {ks:Key ( ∃ t:Terminal_Num
        (ks = Encrypt(KMH0, Session_Key(t))))}
   ∪ {kt:Key ( ∃ t:Terminal_Num
      (kt = Encrypt(KMH1,Terminal_Key(t))))}
 & ∀ k1,k2:Key (k1 ∈ Intruder_Info & k2 ∈ Keys_Used
        → k1 ≠ k2)

   Transform ECPH(K:Key, T:Text) EXTERNAL
   Effect
      N"Intruder_Info =
        ( if T ∈ Intruder_Info & K ∈ Intruder_Info
            then Intruder_Info ∪
              {Encrypt(Decrypt (KMH0,K),T)}
            else Intruder_Info )

   Transform DCPH(K1:Key, T1:Text) EXTERNAL
   Effect
      N"Intruder_Info =
        ( if T1 ∈ Intruder_Info &K1 ∈ Intruder_Info
            then Intruder_Info ∪
              {Decrypt(Decrypt (KMH0,K1),T1)}
            else Intruder_Info )
```

```
Transform RFMK(K1:Key, K2:Key) EXTERNAL
Effect
  N"Intruder_Info =
    ( if K1 ∈ Intruder_Info & K2 ∈ Intruder_Info
      then Intruder_Info ∪ {Encrypt(Decrypt
        (KMH1,K1), Decrypt(KMH0,K2))}
      else Intruder_Info )

Transform Generate_Session_Key(Ter:Terminal_
  Num) EXTERNAL
Effect
  ∃ k:Key ∀ t:Terminal_Num
      (Encrypt(KMH0,k) ∉ Keys_Used
    & Encrypt(Terminal_Key(Ter),k) ∉
        Keys_Used
    & k ∉ Keys_Used
    & N"Session_Key(t) =
      ( if t = Ter
        then k
        else Session_Key(t) )
    & N"Keys_Used = Keys_Used ∪ {k}
    & N"Intruder_Info = Intruder_Info ∪
        {Encrypt(KMH0,k),Encrypt(Terminal
          _Key(Ter),k))}

END Top_Level
END Crypto
```

References

[1] R. V. Book and F. Otto, "The verifiability of two-party protocols," in *Advances in Cryptography—Eurocrypt '85*, 1985.

[2] E. F. Brickell, "Breaking iterated knapsacks," in *Advances in Cryptology: Proceedings of Crypto '84* (Lecture Notes in Computer Science 196). New York: Springer-Verlag, 1985.

[3] E. F. Brickell, J. H. Moore, and M. R. Purtill, "Structure of the S-boxes of the DES," in *Proc. CRYPTO '86*, Santa Barbara, CA, Aug. 1986.

[4] E. F. Brickell and A. M. Odlyzko, "Cryptanalysis: A survey of recent results," *Proc. IEEE*, vol. 76, May 1988.

[5] D. Britton, "Formal verification of a secure network with end-to-end encryption," in *Proc. IEEE Symp. Security Privacy*, Oakland, CA, Apr. 1984.

[6] J. Crow, D. Denning, P. Ladkin, M. Melliar-Smith, J. Rushby, R. Schwartz, R. Shostak, and F. von Henke, "SRI verification system version 2.0 specification language description," Computer Science Lab., SRI International, Menlo Park, CA, Nov. 1985.

[7] D. W. Davies, "Some regular properties of the 'Data Encryption Standard' Algorithm," in *Proc. Crypto '81, Advances in Cryptography*, Dep. Elec. Comput. Eng., Univ. Calif., Santa Barbara, Rep. ECE 82-04, Aug. 1981.

[8] R. A. DeMillo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," *Commun. ACM*, vol. 22, no. 5, May 1979.

[9] "Department of Defense trusted computer systems evaluation criteria," National Computer Security Center, DOD 5200.28-STD, Dec. 1985.

[10] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Inform. Theory*, vol. IT-29, Mar. 1983. An extended abstract appears in *Proc. 22nd IEEE Symp. Foundations Comput. Sci.*, 1981.

[11] S. T. Eckmann and R. A. Kemmerer, "INATEST: An interactive environment for testing formal specifications," presented at the 3rd Workshop Formal Verification, Pajaro Dunes, CA, Feb. 1985; also, *ACM Software Eng. Notes*, vol. 10, no. 4, Aug. 1985.

[12] D. I. Good, B. L. DiVito, and M. K. Smith, "Using the Gypsy methodology," Inst. Comput. Sci., Univ. Texas, Austin, TX, June 1984.

[13] G. J. Holzmann, "Automated protocol validation in Argos: Assertion proving and scatter searching," *IEEE Trans. Software Eng.*, vol. SE-13, June 1987.

[14] C. A. Sunshine, Special Issue on Protocol Specification and Verification, *IEEE Trans. Commun.*, vol. COM-30, Dec. 1982.

[15] R. A. Kemmerer, "Testing formal specifications to detect design errors," *IEEE Trans. Software Eng.*, vol. SE-11, Jan. 1985.

[16] R. A. Kemmerer, *Verification Assessment Study Final Report, Vols. I-V*, National Computer Security Center, Rep. C3-CR01-86, Mar. 1986.

[17] S. S. Lam and A. U. Shankar, "Protocol verification via projections," *IEEE Trans. Software Eng.*, vol. SE-10, July 1984.

[18] D. Longley, "Expert systems applied to the analysis of key management schemes," in *Computers & Security, Vol. 6*. New York: Elsevier Science, 1987.

[19] C. H. Meyer and S. M. Matyas, *Cryptography*. New York: Wiley, 1980.

[20] J. K. Millen, S. C. Clark, and S. B. Freedman, "The Interrogator: Protocol security analysis," *IEEE Trans. Software Eng.*, vol. SE-13, Feb. 1987.

[21] J. H. Moore, "Protocol failures in crypto systems," *Proc. IEEE*, vol. 76, May 1988.

[22] L. Robinson, *The HDM Handbook, Vol I: The Foundations of HDM*, Computer Science Lab., SRI International, Menlo Park, CA, June 1979.

[23] H. Rudin and C. H. West, Eds., *Protocol Specification, Testing, and Verification III*. Amsterdam, The Netherlands: Elsevier Science, North-Holland, 1983.

[24] J. Scheid and S. Holtsberg, *Ina Jo Specification Language Reference Manual*, System Development Group, Unisys Corp., Santa Monica, CA, Sept. 1988.

[25] G. J. Simmons, "How to (selectively) broadcast a secret," in *Proc. IEEE Symp. Security Privacy*, Oakland, CA, Apr. 1985.

[26] C. A. Sunshine, D. H. Thompson, R. W. Erickson, and S. L. Gerhart, "Specification and verification of communication protocols in AFFIRM using state transition models," *IEEE Trans. Software Eng.*, vol. SE-8, Sept. 1982.

[27] D. H. Thompson and R. W. Erickson, Eds., *Affirm Reference Manual*, Inform. Sci. Inst., Univ. Southern Calif., Marina del Rey, CA, Feb. 1981.

[28] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, Aug. 1984.

[29] J. M. Wing and M. Nixon, "Extending Ina Jo with temporal logic," *IEEE Trans. Software Eng.*, vol. SE-13, Feb. 1987.

Richard A. Kemmerer (M'80–SM'89) was born in Allentown, PA, in 1943. He received the B.S. degree in mathematics from The Pennsylvania State University, University Park, in 1966 and the M.S. and Ph.D. degrees in computer science from the University of California, Los Angeles (UCLA), in 1976 and 1979, respectively.

Since 1979 he has been on the Faculty of the Department of Computer Science at the University of California, Santa Barbara, where he is currently an Associate Professor. He has been a visitor at the Massachusetts Institute of Technology, the Wang Institute, and the Politecnico di Milano. From 1966 to 1974 he worked as a programmer and systems consultant for North American Rockwell and the Institute of Transportation and Traffic Engineering at UCLA. His research interests include formal specification and verification, reliable software, and secure systems. He is the author of the book *Formal Specification and Verification of an Operating System Security Kernel*.

Dr. Kemmerer is a Senior Member of the IEEE Computer Society, the Association for Computing Machinery (ACM), and the International Association for Cryptologic Research. He is a past Chairman of the IEEE Technical Committee on Security and Privacy and a member of the Advisory Board for the ACM's Special Interest Group on Security, Audit, and Control.