On Searching for Known and Chosen Cipher Pairs Using the NRL Protocol Analyzer

Stuart G. Stubblebine [*]
AT&T Labs–Research

Catherine A. Meadows [†]
Naval Research Laboratory

Abstract

Formal methods have been successfully applied to exceedingly abstract system specifications to verify high level security properties such as authentication, key exchange, and fail-safe revocation. Furthermore, considerable research exists on evaluating particular ciphers and secure hash functions used to implement high level security properties. However, verifying that less abstract system specifications satisfy low level security properties has been largely impractical. This is evidenced by innumerable system vulnerabilities where high level properties are not attained due to failed assumptions of low level properties. This paper presents ongoing work on investigating known and chosen ciphertext pairs using the NRL Protocol Analyzer. We give a formal characterization of known and chosen pairs, and map it to the NRL Protocol Analyzer model. We also describe the use of the Analyzer to rediscover attacks on an early version of the ESP protocol, and show how our experience in using it has led us to refine our model. This was the first use of the Analyzer to model protocols at such a low level of abstraction.

1 Background

A chosen text attack is characterized by an adversary causing another principal to encrypt or decrypt chosen text using a secret key unknown to the adversary. A known text attack is characterized by an adversary learning encrypted or decrypted text not necessarily chosen by the adversary. Chosen- and known-text in cryptographically protected systems is an important area of study since its presence can lead to vulnerabilities. Chosen- ciphertext can enable the adversary to read confidential messages. Chosen- plaintext can enable the adversary to create messages that may be accepted as legitimate. It can also be used by the adversary to conduct a dictionary attack in an attempt to learn secrets. The problem is compounded when the same key is used for different protocols. Guidance on how to avoid chosen- and known- text attacks has been in the literature for many years [11]. However, attacks based on chosen- and known- text continue to be a problem even in security protocols designed and reviewed by teams of security experts. In spite of this, little work has been done on developing formal theories and techniques for detection and prevention of chosen and known text attacks. This is mainly because the degree of abstraction at which the problem occurs is at a somewhat lower level than that usually handled by most of the existing applications of formal methods of cryptographic protocols. However, as the work of Stubblebine and Gligor [9, 10] has shown us, it is possible to apply formal techniques even at the level at which splicing and message decomposition occur and obtain meaningful results.

Analysis of chosen- and known- text is difficult since analysis is tedious and complex. There are many details to consider including numerous message and protocol options. Another factor making analysis difficult is the size of the search space and the fact that system entities can send anything that is known to it.

In this paper we give a formal characterization of chosen and known text. We also give a theory and method for system specification and analysis for chosen and known text. We present some automated techniques using and available tool, the NRL Protocol Analyzer [6]. We discuss the relationship between these techniques and the theory. Finally, we give some examples using our technique to re-discover known attacks.

*AT&T Labs–Research, Rm B235, 180 Park Ave., Florham Park, NJ 07974, USA. Email: stubblebine@research.att.com
†Naval Research Laboratory, Code 5543, Washington DC 20375, USA. Email: meadows@itd.nrl.navy.mil

2 Definitions

2.1 Block Encryption and Pairs

Definition 1 An l-bit *block encryption* is a function $e : P_l \times \mathcal{K}_m \rightarrow C_l$, such that for each m-bit key $k \in \mathcal{K}_m$ and block $p \in P_l$, $e_k(p)$ has an invertible mapping. The inverse mapping is *block decryption*, denoted $e_k^{-1}(c)$.

Definition 2 A *plaintext-ciphertext pair* is an ordered 2-tuple $\{\langle p, c\rangle \mid \exists k \text{ s.t. } c = e_k(p) \wedge p = e_k^{-1}(c)\}$.

Definition 3 A *plaintext-ciphertext pair under key, k,* is an ordered 3-tuple $\{\langle p, c, k\rangle \mid c = e_k(p) \wedge p = e_k^{-1}(c)\}$.

We will use the term *pair* to refer to a plaintext-ciphertext pair.

2.2 Cipher Modes

We define block cipher modes in terms of block encryption and decryption.

Definition 4 *CBC Mode.*
1. *Encryption.*
   Input: m-bit key k; l-bit IV; l-bit plaintext blocks $p_1, \ldots p_r$.
   Output: $c_0, \ldots, c_r$ *such that* $c_0 \leftarrow IV$ *and* $c_i \leftarrow e_k(c_{i-1} \oplus p_i)$ *for* $1 \leq i \leq r$.
2. *Decryption.*
   Input: m-bit key k; l-bit IV; l-bit ciphertext blocks $c_1, \ldots c_r$.
   Output: $p_0, \ldots, p_r$ *such that* $p_0 \leftarrow IV$ *and* $p_i \leftarrow c_{i-1} \oplus e_k^{-1}(c_i)$ *for* $1 \leq i \leq r$.

We will commonly use the notation $ecbc_k(x, y)$ to denote an encrypted block in a cipher block chain, where x is the previous block of cipher text, y is the current block of plaintext, and k is the key. Likewise, we will use $ecbc_k^{-1}(x, y)$ to denote the result of decrypting a block in a cipher block chain, where x is the previous block of cipher text and y is the current block of cipher text.

2.3 Cipher Block Chaining and Pairs

2.4 Known Pairs

For simplicity, we assume the encryption algorithm may be known.

Definition 5 A pair, $\langle p, c\rangle$, is a *known plaintext-ciphertext pair* with respect to a principal, I, and key, k if and only if I *knows* $\langle p, c\rangle \ \wedge \ \langle p, c, k\rangle$.

We will use the term *known pair* to refer to a known plaintext-ciphertext pair. A principal may know that two pairs are encrypted under the same key even though the key is not known to the principal. For a particular block encryption function, it may be the case that $\langle p, c\rangle$ for any p and c. However, in our analysis, it is of interest to know this fact in the context of receiving particular c and associating it with sets of under the same k. Knowledge of a pair does not imply knowledge of the corresponding key. That is knowing $c = e_k(p)$ or $p = e_k^{-1}(c)$ does not imply knowing k.

2.5 Chosen Pair

Definition 6 A pair, $\langle p, c\rangle$, is a *chosen plaintext pair* with respect to a principal, I, and key, k, if and only if I *choose* $p \wedge I$ *knows* $\langle p, c\rangle \wedge \langle p, c, k\rangle$.

A chosen plaintext pair is a pair whose plaintext can be chosen by a certain principal. Chosen-plaintext pairs can lead to vulnerabilities that enable the adversary to forge messages using the key, read messages, or learn the key. Messages can be forged by exercising the system to create new messages. The key can be learned by precomputing a dictionary of a chosen plaintext under the key space. Note, however, it must be possible for the adversary to obtain the corresponding ciphertext.

Definition 7 A pair, $\langle p, c\rangle$, is a *chosen ciphertext pair* with respect to a principal, I, and key, k, if and only if I *choose* $c \wedge I$ *knows* $\langle p, c\rangle \wedge \langle p, c, k\rangle$.

A chosen ciphertext pair is a pair whose ciphertext can be chosen by a certain principal.

3 Analysis Using the Analyzer

We begin by given a brief introduction to the NRL Protocol Analyzer. We then compare the rules of the analyzer with the rules outlined in the previous section. We go on to augment the analyzer rules to capture the necessary formalism. We give informal arguments on the mapping between our definitions and analysis using the analyzer.

3.1 Overview of the Analyzer

The NRL Protocol Analyzer is a formal methods tool for analyzing security properties of cryptographic protocols. Protocols are specified as communicating state machines, one of which is a hostile intruder who can read all traffic, modify or delete traffic, perform cryptographic operations, and may be in cooperation with some legitimate users of the system. The intruder also is assumed to have knowledge about relationships between data. That is, if it sees the result of encrypting a piece of plaintext with a key, it will know that it is the result of encrypting the plaintext with that key, although it may not know plaintext or key. This model has the effect of simplifying our search for known pairs, since it means that if the intruder sees a plaintext-ciphertext pair, it also knows that it is such.

The user of the Analyzer attempts to determine whether or not a protocol is secure by specifying an insecure state. The Analyzer works backwards from that state until it either reaches an initial state, in which case it has found an attack, or until it has shown that all paths begin in an unreachable state.

The Analyzer makes no assumptions about the limits on the number of protocol executions, the number of principals performing the different executions, the number of interleaved executions, or the number of times cryptographic functions are applied. This results in a search space that is originally infinite. However, the Analyzer provides means for specifying and proving inductive lemmas about the unreachability of infinite classes of states. This allows the user to narrow down the search space so that in many cases an exhaustive search is possible.

An Analyzer specification consists of four parts. The first part describes the operations and terms used in the specification. The second part describes the rewrite rules used. These capture the necessary algebraic properties of cryptosystems, such as the fact that encryption and decryption cancel each other out. The third part specifies the words known initially by the intruder. The last and main part consists of transition rules describing the actions taken by legitimate participants. The inputs to these transition rules are messages received (which may have been passed on by the intruder) and values of local state variables. The outputs are messages sent (which will be available to the intruder) and new values of the local state variables. The intruder actions are not specified directly, but are generated automatically from the specification.

The Analyzer generates the input state to an existing state defined as a set of local state variables values and words known to the intruder as follows. It takes each transition rule output in turn and uses a narrowing algorithm to find a complete description of all substitutions that make any portion of the output reducible to a subset of the state. The input to the rule and the unmatched portion of the state then become the input state. This allows the Analyzer to discover unexpected consequences of the use of cryptographic functions.

The Analyzer has been applied to a number of different cryptographic protocols, and has found flaws in several. In some cases the flaws had not been discovered before. Examples of protocols the Analyzer has been used to examine are the Simmons Selective Broadcast Protocol [5], the Burns-Mitchell Ticket Granting Protocol [4], the Needham Schroeder public key protocol [7]. A description of the Analyzer is given in [6].

Most of the protocols to which the Analyzer has been applied are specified at a higher degree of abstraction than the protocols we will be looking at in this paper. Methods for using block ciphers to encrypt messages more than one block long are not modeled, and the integrity of an encrypted message is assumed. Thus, using the Protocol Analyzer to explore the consequences of cipher block chaining was a new departure.

3.2 Notation and Rewrite Rules

Notation The following notation applies for mapping between the analyzer and our formalism. This will help us in our discuss of queries in the next session.

1. $ecbc_k(x, y) \equiv$ `ecbc(k, x, y)`
2. $ecbc_k^{-1}(x, y) \equiv$ `dcbc(k, x, y)`

Rewrite Rules Here we give a set of rewrite rules that describe the relationship between encryption and decryption. These are the rules that will be used by the Analyzer to model the effects of encryption and decryption.

```
ecbc(K,IV,dcbc(K,IV,C))→ C
dcbc(K,IV,ecbc(K,IV,P))→ P
```

3.3 Queries for Finding Known and Chosen Pairs

Claim 1 *If a search for the intruder knowing* `ecbc(K,C,P), C` *and* `P` *has a solution, then there exists known pairs under the key, K.*

This follows immediately from our assumption that the intruder always knows relations between words if they exist, and from the fact that, if the intruder knows `C` and `P`, then the intruder can compute exclusive-or of `C` and `P`. The converse of this statement is of course not true; it may be possible that the intruder can learn the exclusive-or of `C` and `P` without knowing `C` and `P`. However, if we are able to rule out knowledge of `C` and `P`, we will have been able to rule out one important way of finding known pairs.

The next two claims describe the intruder's ability to produce chosen plaintext and ciphertext. We make use of a function called `notsent`. The term `notsent(X)` can be computed by the intruder if and only if it already knows `X`. We guarantee the "if" part of the above claim by specifying `notsent(X)` as a function computable by the intruder, and the "only if" part by having all messages generated by an honest principal (in this case, a host) contain as subterms either constants, timestamps, function symbols other than `notsent`, or subterms of earlier messages received. We can verify that these restrictions guarantee that `notsent(X)` is achievable only if the intruder knows `X` by giving the Analyzer as a goal the state in which the intruder knows `notsent(X)` but has not computed `notsent(X)` from `X`. We then prove that this goal state is unreachable.

The reason for choosing this definition of `notsent` is that we want to be able to model an arbitrary function that the intruder can use to produce chosen plaintext when CBC mode is used. The idea is as follows. Suppose that the intruder can produce `ecbc(K,IV,notsent(IV))`. Then the intruder can choose `notsent(IV)` to be the exclusive-or of `IV` and a chosen plaintext `P`. Applying the cipher block chaining algorithm to `IV` and `notsent(IV)` would result in $e_K(P)$ which would give us the chosen plaintext we would need. A similar argument would produce chosen ciphertext.

Given this motivation, our claims are as follows:

Claim 2 *If a search for the intruder knowing* `ecbc(K,notsent(P),P)`*, or* `ecbc(K,IV,notsent(IV))` *has a solution, then there exist chosen-plaintext pairs under the key, K.*

Claim 3 *If a search for the intruder knowing* `dcbc(K,notsent(C),C)`*, or* `dcbc(K,IV,notsent(IV))` *has a solution, then there exist chosen-ciphertext pairs under the key, K.*

We note again that we have no guarantee that the converse of these statements is true. Knowing that intruder can't produce `ecbc(K,notsent(P),P)` or `ecbc(K,IV,notsent(IV))` does not eliminate the possibility that the intruder may be able to force the choice of `X` and `Y` so that their exclusive-or is a chosen `P`, without being able to choose `X` or `Y` directly. However, we do know that if we can rule out the intruder's finding `ecbc(K,notsent(P),P)` and `ecbc(K,IV,notsent(IV))`, we can rule out one of the most likely ways the intruder has of finding chosen plaintext.

Our claims in this section replaces a previous set of claims which relied upon the intruder's use of a constant `notsent` instead of the function `notsent(P)`. While this was sufficient to capture known and chosen pairs for Electronic Code Book Mode, it was not sufficient to capture the fact that the intruder needed knowledge of one of the arguments of `ecbc` or `dcbc` before he computed the other. The incorrect use of `notsent` as a constant led to the discover of spurious "attacks" on protocols. In the next section, we will describe show how our analysis of the IP Encapsulating Security Protocol led to a discovery of such a spurious attack. We will also show how the discovery of these spurious attacks helped us to identify an error both in our specification and in our requirements.

3.4 Analysis of the IP Encapsulating Security Protocol

In this section we describe the current state of our analysis of an early version of the IP Encapsulating Security Protocol or ESP [1, 8]. ESP is the protocol of the Internet security architecture for securing IP [2]. It describes formats and transforms for encryption and decryption of data. We selected ESP because its use of CBC mode caused Bellovin to notice a number of possible attacks when it was used under certain system configurations [1, 8].

As we mentioned in the last section, our specification of the protocol has an error in it. However, we describe it here both because even with the error we were able to reproduce a number of Bellovin's attacks, and also because we wish to document the process by which the discovery of the error led us to revise our formal definitions of chosen pairs.

3.4.1 Description of the Attacks

In Bellovin's scenario, secure communication is between hosts. Each pair of hosts shares a key, which is not changed between sessions. Each host has a number of users communicating with it, some of whom are honest, and some of whom are actively trying to subvert the protocol. Cipher block chaining is used to encrypt packets, and IVs are sent in the clear. Each packet contains an unencrypted ESP header, containing a SPI (Security Parameter Index) which corresponds to a key shared between hosts. SPIs and their corresponding keys are one-way, that is, if host A initiates contact with host B, it uses a different SPI than if host B initiates contact with host A. Encrypted packets contain headers that include such information as to who sent the message and for whom it is intended. Depending upon the communication protocol used, different types of header formats may be used.

There are at least two ways in which a dishonest user could subvert the protocol:

a. He could pass a message from himself as coming from an honest user U by appending a fake header H1 identifying the message as coming from U to its beginning and giving it to the host on which U resides to encrypt. The host would append another header H2 to the message and encrypt it. The dishonest user could then truncate the encrypted message at H1, and pass it off as a message with header H1, using the last block of the encrypted H2 as the IV. Bellovin describes a similar attack in [3].

b. If the intruder resides at host A, she could learn a message M1 intended for another user U at the same host A as follows. First she would need to find out the host B originating the message. She would take another message M2 from B to A intended for herself, and remove the encrypted header EH2. She would then append the last portion M1' of M1 to EH, so that it is of the expected length. The host would then decrypt the entire message, and return the decrypted M1' to the intruder. Part of M1' could be garbled, but thanks to the self-healing properties of cipher block chaining, the remainder would be readable. Bellovin describes this attack in [3].

3.4.2 The Specification

In order to see whether the Analyzer could find attacks like this, we modeled a protocol along the line of Bellovin's scenario. In order to make the analysis tractable, we assumed packets were only four blocks long. We modeled two types of headers. One was two blocks long, and one was only one block. This meant that message bodies were always at least two blocks long, which meant that we were able to illustrate the self-healing properties of cipher block chaining; even if the first block was garbled, the second would be correct. We also assumed that hosts would only continue decrypting a packet if the header parsed; this allowed us to discard cases in which header was garbled, and hence a message could neither be passed to the intruder or regarded as a legitimate message for an honest user. This allowed us to ignore a number of useless states that would not have aided us in the analysis but would have increased the size of the search space. We also assumed that SPIs were simplex, so that host A encrypting a message for host B always used `spi(A,B)` when encrypting a message, and `spi(B,A)` when decrypting it.

We made a severely simplifying assumption about headers. We assumed that the only thing distinguishing one header from another was its type (short or long) and who the senders and receivers of the message were. This was probably oversimplification, and may have prevented us from finding some subtle attacks in which different sessions between the same parties were confused. However, we believed that it was enough for our goal of determining whether sessions between different parties could be confused. We also assumed that the intruder knows all headers.

Our specification of cipher-block chaining is recursive. When encrypting, the host stores the last encrypted block it produced in a local state variable. It then applies the cipher-block chaining algorithm to the last encrypted block and the next plaintext block. It then replaces the last encrypted block in the state variable with the new encrypted block, which it will use to produce the next encrypted block. Plaintext blocks are either produced by the host (the case when the producer of the message is an honest user) or supplied by the intruder (the case when the producer of the message is a dishonest user). Our reason for having the host produce the messages for honest users is that we assume that communication between the honest user and the host is secure, and that the honest user and host obey the same policy. Thus we may identify the honest user with the host. Likewise, since we assume that all dishonest users are in cooperation with the intruder, we may assume that all messages from dishonest users are produced by the intruder.

Our specification of decryption is similar. The main difference is that all ciphertext messages are assumed to be received from the intruder, since they are passed along an insecure network. The host stores the last ciphertext block received and applies the cipher block chaining decryption algorithm to the next block received. Once it has produced the decrypted block, it replaces the last ciphertext block with the one it just received, and awaits the next ciphertext block.

Our recursive definition of cipher block chaining led to a certain amount of explosion in the state space, since the Analyzer attempted to interleave encryption and decryption transitions. Although the Analyzer has some facilities for recognizing and avoiding multiple interleavings of events, this did not prevent them all. However, even with the state explosion we were able to find Bellovin's attacks. However, our use of a recursive specification style made the specification much easier to read and write. It also made it possible to model a host's refusal to proceed with decryption if it could not read the header, which in itself was a significant help in reducing the search space. However, in order to analyze larger protocols, we will probably need better ways of cutting down on the state explosion.

Our specification also contained an inaccuracy that could lead to the discovery of spurious attacks. In order to simplify the recursive specification, we assumed that blocks were sent out to the network as soon as they were encrypted or decrypted. This of course is not the case; blocks are not sent until the entire packet is encrypted or decrypted. Our "simplifying" assumption meant we discovered "attacks" in which the intruder used information about one block to influence the production of the next block. We will discuss this in more detail in the next section.

Our specification contained twelve rules:

1. A rule describing a host encrypting the first part of a header for a message from an honest user;

2. A rule describing a host encrypting the second part of a long header for a message from an honest user;

3. A rule describing a host encrypting the rest of a message for an honest user;

4. A rule describing a host encrypting the first part of a header for a dishonest user;
5. A rule describing a host encrypting the second part of a long header for a message from a dishonest user;
6. A rule describing a host encrypting the rest of a message for a dishonest user;
7. A rule describing a host decrypting the first part of a header;
8. A rule describing a host decrypting the second part of a long header;
9. A rule describing a host decrypting the first block appearing after a long header (this needed to be included to model the host's refusal to proceed if the header had not decrypted properly);
10. A rule describing a host decrypting the first block appearing after a short header ;
11. A rule describing a host decrypting the remaining blocks in a message, and;
12. A rule describing a host revealing a decrypted block to the intruder if the header indicates that the message is intended for a dishonest user.

A copy of the specification is included in the Appendix.

3.4.3 Results of Using the Analyzer

We posed four questions to the Analyzer. In the first two, we asked it whether or not if could find `X` and `ecbc(K,X,notsent)` or `X` and `dcbc(K,X,notsent)`, using our initial model of chosen pairs for cipher block chaining. The Analyzer found these attacks easily in a few seconds. For chosen ciphertext, the Analyzer found an attack in four steps.

1. Host A encrypts message header from honest user U at to dishonest user V at Host B.
2. V sends IV and encrypted header EH on to B.
3. V appends notsent to EH as the next part of the encrypted message. This is also sent to B.
4. B decrypts message, and returns dcbc(K,X,notsent) to the intruder, where X is the last part of the encrypted header. The intruder of course has already learned this from Step 2.

The Analyzer also produced the trivial chosen plaintext attack: The intruder includes `notsent` as part of a message to be encrypted. Since the input `X` to `ecbc(K,X,notsent)` is just the previous encrypted block of the message, the intruder is easily able to produce `X` and `ecbc(K,X,notsent)` in this way.

This was the spurious attack. In the actual ESP protocol, the attacker would see `X`, but would not see it before it sent `notsent` to the to the host to be encrypted. Thus it would not be able use its knowledge of `X` to influence its choice of `notsent`. Realization of our mistake lead not only to a revision of the specification, which is in progress, but to the revision of our definition of chosen pairs that we described in Section 3.3.

The Analyzer also found a number of attacks similar to Bellovin's. We generated a spoofing attack by asking the Analyzer if it could find a state in which Host A could enter a state in which its value for the originator of a message was an honest user U2, but the value for the decrypted block was `notsent`. It returned, among other things, the following path:

1,2,3. Host A encrypts a message M1 from dishonest user V to some user U1 at host B. The first part of the message is a header for a message M2 from an honest user U2. The rest is the spoofed message the intruder wants to send. After encrypting, the intruder removes the encrypted message header.

V sends the three encrypted blocks to B, passing off the first block as the IV. A fourth random block could be included to make up the entire packet, although this step was not included by the Analyzer.

4. B decrypts the apparent first part of the message, and accepts that as the header.

5. B decrypts the apparent rest of the message, and accepts that as the message from E.

Bellovin describes a similar attack in [3]. In his attack a a legitimate message M2 is sent from U1 to U2. The intruder constructs a message M1, and then cuts off the portions after the headers on M1 and M2. The last portion of M1 is appended to the first part of M2. This attack is somewhat stronger than ours, since it allows an intruder to hijack a session without necessarily knowing the header associated with that session. We expect that we can also produce this attack as we continue our search.

The Analyzer found an unauthorized disclosure attack in seven steps. We produced this by representing a block of a message produced to be sent from `user(U1,honest)` at host A to `user(U2,honest)` at host B as `message(user(U1,honest),user(U2,honest),ts(host(A),N),Num)` where `Num` indicates the block's position in the packet and `ts(host(A),N)` is a host timestamp guaranteeing the uniqueness of the message. The following path of attack was found:

1,2. Honest user U1 at Host A encrypts a message to another honest user U2 at Host B. This takes two steps in the Analyzer, one to encrypt the header, and one to encrypt the rest of the message (actually, the first word of the rest of the message; this was all the Analyzer needed to find the attack). The result of this is IV1, EncryptedHeader1, Message1.

3. Another user U3 at Host A encrypts a message to dishonest user V at Host B. In this case, all we are interested in is the header, so this only takes one step. The result of this is IV2, Encryptedheader2.

   The intruder sends the message IV2, EncryptedHeader2, EncryptedHeader1, Message1 to Host B.

4. Host B decrypts the header, and concludes that the remainder of the message will be for the dishonest user V.

5. Host B uses EncryptedHeader 2 to decrypt Encrypted Header1, which is garbage.

6. Host B uses EncryptedHeader1 to decrypt Message1.

7. Host B passes Message1 to the intruder.

Note that the garbage decryption of EncryptedHeader1 is also passed on to the intruder, but, since this does not aid in the attack, it does not appear in the search.

This is exactly the unauthorized disclosure attack found by Bellovin.

4 Conclusion

This work constitutes a significant step towards proving secure uses of mechanisms that assume absence of known or chosen pairs. We have given a characterization of known and chosen pairs, and a means of specifying their presence or absence. We have also shown how it can be applied to the use of an existing tool, the NRL Protocol Analyzer, and how the tool could use the model to find ways of generating chosen pairs in the ESP protocol. We then went on to show how it could be extended to be used to find attacks based on cutting and pasting.

Much work still remains to be done. In work connected to this paper, we have developed a number of inference rules describing the conclusions an attacker may draw about known and chosen pairs. As yet these have no formal foundation. However, we have begun to map these rules to the process that can be taken by the NRL Protocol Analyzer. We see this as a way, not only of providing a more concrete basis for our inference rules, but also possibly a means of abstracting away from the more computationally intensive approach of the Analyzer.

We also need to consider strategies for developing specifications for the Analyzer. The assumptions we made (that all packets were four blocks long, that headers took up one or two blocks entirely) were designed with the idea in mind of capturing the relevant security features of the specification, while still keeping the analysis tractable. However, our approach to doing this was informal and ad hoc, and as we have shown, led to some errors that resulted in the generation of spurious attacks. We need to develop more systematic ways of identifying assumptions, as well as rigorous arguments that our abstractions capture the assumptions.

This will give us ways of generating specifications in a more systematic fashion, and will allow us more confidence in the usefulness of any proofs that we obtain that a specification is secure against certain types of attacks. However, as we learned from writing the specification that appears in this paper, it is possible to clarify assumptions and discover errors by examining the attacks produced by the Analyzer and comparing them against what we know to be possible. We expect continued use of the Analyzer to test our hypotheses to help us further in the development of our heuristics.

Finally, there are probably some improvements that need to be made to the Analyzer itself before it can be used successfully to analyze more complex protocols involving low-level properties. Surprisingly, we did not find the Analyzer's inability to model the commutivity and associativity of exclusive-or as much of a handicap as we had feared. This was because many threats posed by cipher block chaining, that is, its cut-and-paste and self-healing properties, were easily expressible in terms that could be modeled by the Analyzer. However, we found that the specification of block-by-block encryption was complex. Consequently, the analysis resulted in a certain amount of state space explosion. This could clearly be a drawback if we wanted to specify anything more than the simple sending and receiving of messages. However, we note that, as part of the work on the Protocol Analyzer this year, the second author of this paper has developed a more sophisticated specification language for the Protocol Analyzer that makes for much cleaner specifications. Indeed, this language had been used to specify the ISAKMP/Oakley protocol and the extremely complex SET protocol. Since the language incorporates constructs such as subroutines and if-then-else, it should be easier to modify recursive procedures in it.

The problem of state explosion is tougher. However, we note that much of the state explosion results from the recursive specification generating a number of very similar-looking states. The Analyzer already includes some features for recognizing when one state is reachable only if another state is has a way of cutting down on the size of the search space. It may be possible to extend this to permit the Analyzer to perform induction over recursively generated states.

References

[1] R. Atkinson. IP encapsulating security payload (ESP). Request for Comments (Proposed Standard) RFC 1827, Internet Engineering Task Force, August 1995.

[2] R. Atkinson. Security architecture for the internet protocol. Request for Comments (Proposed Standard) RFC 1825, Internet Engineering Task Force, August 1995.

[3] S. M. Bellovin, Problem Areas for the IP Security Protocols, *Proceedings of the Sixth Usenix UNIX Security Symposium*, San Jose, CA, July, 1996.

[4] C. Meadows, A System for the Specification and Verification of Key Management Protocols, *Proceedings of the 1991 IEEE Computer Society Symposium in Research in Security and Privacy*, May 1991.

[5] C. Meadows, Applying Formal Methods to the Analysis of a Key Management Protocol, *The Journal of Computer Security*, 1992:1,1, Jan, 1992.

[6] C. Meadows, The NRL Protocol Analyzer: An Overview, *J. Logic Programming*, 1996:19, 20:1-679, Elsevier Science, New York, NY, 1996.

[7] Catherine Meadows, Analyzing the Needham-Schroeder Public-Key Protocol: A Comparison of Two Approaches, *Computer Security - ESORICS 96*, Springer-Verlag, September 1996, pp. 365-364.

[8] P. Metzger, P. Karn, and W. Simpson. The ESP DES-CBC transform. Request for Comments (Proposed Standard) RFC 1829, Internet Engineering Task Force, August 1995.

[9] S. Stubblebine and V. Gligor, Protocol Design for Integrity Protection, *IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May, 1993, pp. 41-53.

[10] S. Stubblebine and V. Gligor, On Message Integrity in Cryptographic Protocols, *IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May, 1992, pp. 85-104.

[11] V. L. Voydock and S. T. Kent, Security Mechanisms In High- Level Network Protocols, *Computing Surveys*, vol. 15, no. 2, June 1983.

Appendix: Specification of the ESP Protocol

```
fs(1):host(A) --> W::.
fs(2):iv(U,N) --> W::.
fs(3):key(U,V) --> W::.
fs(4):header(U,V,Type,Num) --> W::.
fs(6):mess(U,V,TS,Num) --> W::.
fs(7):plus(Num) --> W::.
fs(8):ts(U,N) --> W::.
fs(9):user(A,H) --> W::.
fs(10):spi(U,V) --> W::.

op(2):ecbc(X,Y,Z) --> W:::pen.
op(3):dcbc(X,Y,Z) --> W:::pen.


atom(1):honest --> W:.
atom(2):dishonest --> W:.
atom(3):notsent --> W:.
atom(4):nil --> W:.
atom(5):one --> W:.
atom(6):short --> W:.
atom(7):long --> W:.
atom(8):word --> W:.
atom(9):header --> W:.


rr(1):ecbc(K,IV,dcbc(K,IV,Z)) => Z.
rr(3):dcbc(K,IV,ecbc(K,IV,Z)) => Z.

known: host(A), user(A,H), header(U,V,Type,Num),
mess(user(C,dishonest),V,TS,Num),
notsent, spi(H1,H2),
plus(Num).

/*host(A) sends message from host user user(C,honest) to user(D,H)
on host(B)
message is
spi(host(A),host(B)), [SA denotes the security association]
header(user(C,honest),user(D,H)) [indicates that the header says
    who the message is from and who it is
    to, among other things]
data(user(C,honest),user(D,H),ts(host(A),N))
   [denotes actual encrypted data sent
   the ts(host(A),N) is a syntactic device to ensure
   uniques of encrypted data (we are assuming that
   encrypted data is unique]*/
```

```
rule(1)
If:
count(host(A)) = [N],
then:
count(host(A)) = [s(N)],
lfact(host(A),N,host_encryptstate,s(N)) =
  [host(B),user(C,honest),user(D,H),
 ecbc(key(host(A),host(B)),
  iv(host(A),N),header(user(C,honest),user(D,H),Htype,one)),
 header,Htype,one],
intruderlearns([spi(host(A),host(B)),iv(host(A),N),
 ecbc(key(host(A),host(B)),
  iv(host(A),N),
  header(user(C,honest),user(D,H),Htype,one))]),

EVENT:
event(host(A),N,host_firstheader,s(N)) = [host(B),user(C,honest),user(D,H),
iv(host(A),N)].

rule(2)
If:
count(host(A)) = [T],
lfact(host(A),N,host_encryptstate,T) = [Host,user(C,honest),V,CT,header,long,Num],
not(Num = plus(Num1)),
then:
count(host(A)) = [s(T)],
lfact(host(A),N,host_encryptstate,s(T)) = [Host,user(C,honest),V,
 ecbc(key(host(A),Host),CT,header(user(C,honest),V,long,plus(Num))),header,
 long,plus(Num)],
intruderlearns([ecbc(key(host(A),Host),CT,header(user(C,honest),V,Htype,plus(Num)))]),
EVENT:
event(host(A),N,host_secondheader,s(T)) = [Host,user(C,honest),V,CT,Num].

rule(3)
If:
count(host(A)) = [T],
lfact(host(A),N,host_encryptstate,T) = [Host,user(C,honest),V,CT,Type,Htype,Num],
not(Num = plus(plus(plus(Num1)))),
then:
count(host(A)) = [s(T)],
lfact(host(A),N,host_encryptstate,s(T)) = [Host,user(C,honest),V,
 ecbc(key(host(A),Host),CT,mess(user(C,honest),V,ts(host(A),T),Num)),
 word,Htype,plus(Num)],
intruderlearns([ecbc(key(host(A),Host),CT,mess(user(C,honest),V,ts(host(A),T),Num))]),
EVENT:
event(host(A),N,host_messageword,s(T)) = [Host,user(C,honest),V,CT,Num].

/*Rules describing intruder encrypting a message*/

rule(4)
```

```
If:
count(host(A)) = [N],
intruderknows([user(C,dishonest),user(D,H)]),
then:
count(host(A)) = [s(N)],
lfact(host(A),N,host_encryptstate,s(N)) =
  [host(B),user(C,dishonest),user(D,H),
 ecbc(key(host(A),host(B)),
  iv(host(A),N),header(user(C,dishonest),user(D,H),Htype,one)),
 header,Htype,one],
intruderlearns([spi(host(A),host(B)),iv(host(A),N),
 ecbc(key(host(A),host(B)),
  iv(host(A),N),
  header(user(C,dishonest),user(D,H),Htype,one))]),
EVENT:
event(host(A),N,host_firstbadguyheader,s(N)) = [host(B),user(C,dishonest),user(D,H),
    iv(host(A),N)].

rule(5)
If:
count(host(A)) = [T],
intruderknows([header(user(C,dishonest),V,Htype,plus(one))]),
lfact(host(A),N,host_encryptstate,T) = [Host,user(C,dishonest),V,CT,header,long,Num],
not(Num = plus(Num1)),
then:
count(host(A)) = [s(T)],
lfact(host(A),N,host_encryptstate,s(T)) = [Host,user(C,dishonest),V,
 ecbc(key(host(A),Host),
    CT,header(user(C,dishonest),V,long,plus(Num))),header,Htype,plus(Num)],
intruderlearns([ecbc(key(host(A),Host),CT,header(user(C,dishonest),V,long,plus(Num)))]),
EVENT:
event(host(A),N,host_badguysecondheader,s(T)) = [Host,user(C,dishonest),V,CT,Num].

rule(6)
If:
count(host(A)) = [T],
intruderknows([X]),
lfact(host(A),N,host_encryptstate,T) = [Host,user(C,dishonest),V,CT,Type,Htype,Num],
not(Num = plus(plus(plus(Num1)))),
then:
count(host(A)) = [s(T)],
lfact(host(A),N,host_encryptstate,s(T)) = [Host,user(C,dishonest),V,
 ecbc(key(host(A),Host),CT,X),
 word,Htype,plus(Num)],
intruderlearns([ecbc(key(host(A),Host),CT,X)]),
EVENT:
event(host(A),N,host_badguymessageword,s(T)) = [Host,user(C,dishonest),V,CT,Num].


/*Host decrypts message*/

rule(7)
If:
```

```
count(host(A)) = [N],
intruderknows([spi(host(B),host(A)),IV,X]),
then:
count(host(A)) = [s(N)],
lfact(host(A),N,host_decryptstate,s(N)) = [host(B),nil,nil,
 X,dcbc(key(host(B),host(A)),IV,X),header,one],
EVENT:
event(host(A),N,host_first,s(N)) = [host(B),IV,X].


rule(8)
If:
count(host(A)) = [T],
intruderknows([X]),
lfact(host(A),N,host_decryptstate,T) = [host(B),nil,nil,
 Y,header(user(C,H),user(D,J),long,one),header,one],
not(Num = plus(Num1)),
then:
count(host(A)) = [s(T)],
lfact(host(A),N,host_decryptstate,s(T)) = [host(B),user(C,H),user(D,J),
 X,dcbc(key(host(B),host(A)),Y,X),header,plus(one)],
EVENT:
event(host(A),N,host_continuedheader,s(T)) = [host(B),user(C,H),
 user(D,J),Y,X].

rule(9)
If:
count(host(A)) = [T],
intruderknows([X]),
lfact(host(A),N,host_decryptstate,T) = [host(B),user(C,H),user(D,J),
 Y,header(user(C,H),user(D,J),long,plus(one)),header,plus(one)],
then:
count(host(A)) = [s(T)],
lfact(host(A),N,host_decryptstate,s(T)) = [host(B),user(C,H),user(D,J),
 X,dcbc(key(host(B),host(A)),Y,X),word,plus(plus(one))],
EVENT:
event(host(A),N,host_decryptedfirstword,s(T)) = [host(B),user(C,H),
 user(D,J),X,Y].

rule(10)
If:
count(host(A)) = [T],
intruderknows([X]),
lfact(host(A),N,host_decryptstate,T) = [host(B),nil,nil,
 Y,header(user(C,H),user(D,J),short,one),header,one],
then:
count(host(A)) = [s(T)],
lfact(host(A),N,host_decryptstate,s(T)) = [host(B),user(C,H),user(D,J),
 X,dcbc(key(host(B),host(A)),Y,X),word,plus(one)],
EVENT:
event(host(A),N,host_decryptedfirstword,s(T)) = [host(B),user(C,H),
 user(D,J),X,Y].
```

```
rule(11)
If:
count(host(A)) = [T],
intruderknows([X]),
lfact(host(A),N,host_decryptstate,T) = [host(B),user(C,H),user(D,J),
 Y,Z,word,Num],
not(Num = plus(plus(plus(Num1)))),
then:
count(host(A)) = [s(T)],
lfact(host(A),N,host_decryptstate,s(T)) = [host(B),user(C,H),user(D,J),
 X,dcbc(key(host(B),host(A)),Y,X),word,plus(Num)],
EVENT:
event(host(A),N,host_decryptedword,s(T)) = [host(B),user(C,H),
 user(D,J),X,Y].
```

```
rule(12)
If:
count(host(A)) = [T],
lfact(host(A),N,host_decryptstate,T) = [host(B),user(C,H),user(D,dishonest),
 X,Y,word,Num],
then:
count(host(A)) = [s(T)],
intruderlearns([Y]),
EVENT:
event(host(A),N,host_revealedword,s(T)) = [host(B),user(C,H),
 user(D,dishonest),X,Y].
```