Abstract—Large scale distributed systems often require security protocols to ensure high integrity. We present a modeling approach that uses UML 2 without extensions to support the design, composition and verification of security protocols. The approach assumes a strong threat model, in which an attacker can intercept, modify, and spoof all communications, with the exception of those protected by known-strong encryption. Through a series of models of extensively-studied protocols we demonstrate that the approach allows protocol properties to be accurately represented, and protocols to be automatically tested to detect potential security flaws. The approach benefits from the existing strong tool support for UML 2, allowing automatic generation of protocol implementations from the models.

Keywords: Security Protocols, UML 2, Visual Modeling, Model Driven Development

I. INTRODUCTION

A typical distributed computer system consists of a number of different participants that range from people, companies, computers, to devices such as magnetic card readers that all must communicate through any number of different channels such as distributed computer networks, phone lines, radio, infrared or devices such as bank cards. The architectural key-stone of any secure computer system is the security protocol that provides the building blocks for ensuring secure electronic communications between the participants. While the design of secure communication protocols has improved over the years the tasks of building and validating these protocols remains inherently difficult. Failures occur due a result of unintended use, malicious attacks, incorrect logic or incorrect transition from design to code.

The vulnerabilities encountered within a secure system are often a result of flaws within the implementation as opposed to flaws in the actual design of the security mechanism [1]. Due to lack of expertise with systems developers, security is not always considered as part of the initial requirements and design but instead is something that is engineered into the software after the design work is complete and the functional requirements have been captured and modeled. This results in the actual definition and integration of the security policy being treated separately and after the fact, meaning that security policy enforcement mechanisms are retro-fitted to pre-existing designs [2]. Given that attacks on systems can pose a serious threat to economic and physical well-being of people and organizations, any effort directed at improving quality in the development of secure systems can aid in reducing the risk and exposure to these types of attacks [3].

While a number of approaches have been developed to address the correct modeling and implementation of security protocols, there is still a significant semantic gap between the design of security protocols in the context of software engineering and the transition from design to implementation. Our approach is to use UML 2 without extensions to support the design, construction and verification of security protocols. The use of UML 2 and Model Driven Development (MDD) to create executable models provides the means to ensure that design intent is directly translated into code and to validate the implementation as the design, as expressed by the model, can be automatically compiled and built into executable code. Within our framework, once the executable code has been created, the protocol can be executed under normal conditions as well as under different threat scenarios.

In the next section, we provide an overview of security protocols and a survey of current approaches to the modeling of security protocols. Section 3 contains a description of our approach to the modeling and validation of security protocols utilizing new constructs within UML 2 and MDD. We then conclude with our findings and a summary of our work.

II. BACKGROUND

A security protocol is a set of rules and conventions that prescribe a sequence of interactions between participants to achieve a specific security goal. A standard notation has been informally adopted by the security protocol community as a means by which to describe a protocol and the sequence of steps between the participants or principals involved in the communications [4]. Principals are identified by a set of common names: Alice, Bob, Server, Adversary, etc. The standard notation describes only the messages that would be passed in a successful run of the protocol. Where there are problems, such as a message of the wrong type being passed or no message received at all, these situations are not described within this notation.

A very simple symmetric protocol in which Alice, Bob and the Server exchange messages in order to establish a key to be used in the communications between Alice and Bob would appear in standard notation as follows:

1) $A \rightarrow S : A, B$
2) $S \rightarrow A : \{K_{AB}\}K_{AS}, \{K_{AB}\}K_{BS}$
3) $A \rightarrow B : \{K_{AB}\}K_{BS}, A$

Messages that are sent in the clear may be read by any principal that has access to the message, but an encrypted message, denoted by $\{\}$ and a key $K_{AS}$, can only be read by those principals that know the key that is required to decrypt the message. Protocols that employ encryption are often referred to as cryptographic protocols, for ease of use within this paper the term security protocol will be used to refer to both types of protocols.

A. Approaches to Modeling Security Protocols

Given that security protocols are a critical component of the security architecture for any distributed system, the correct implementation of these protocols becomes a serious issue. Incorporating security protocols early in the system development process requires a mechanism to support better understanding, design and implementation by systems developers who may not be specialists in security. A number of approaches have been developed to address the correct modeling and implementation of security protocols from formal methods through to custom visual formalisms and extensions to UML.

1) Formal Methods.: Formal methods are mathematically based techniques that can be used to describe the properties of a system that typically involve proofs of correctness. The strength of formal methods is that they provide the framework to construct a provable mathematical model of a system and conduct a formal study of that system in order to reason about security protocols and their properties. While they may be expressive and predictive, they have not been widely adopted by industry given that they require more specialized knowledge and training to effectively apply them. Unfortunately, this makes these types of approaches less cost-effective than other methods as security and software engineers cannot easily learn and apply these methods. In addition, they are not specifically suited to the broader issue of integrating security engineering into software engineering in order to bridge the gap between design and implementation; this gap also extends to design-code verification.

2) Custom Visual Formalisms.: Some solutions present visual modeling formalisms intended to address both the complexity of formal methods and the deficiencies perceived to exist within other modeling languages. An example is the GSPML modeling language, intended as a solution to the security-specific problem of modeling protocols using a visual modeling language. In GSPML a the model represents all possible traces for the protocol, including not only the correct behaviour but also the behaviour of the protocol when under attack [5].

The strength of GSPML is that it uses visual models to represent complex concepts in a concise fashion in order to aid in understanding and verification. While intended for use by security specialists, GSPML is also intended to provide a bridge from the specialist to software developers given its visual presentation approach [5]. However, GSPML is specific only to security protocols, introducing a new language and notation that is still targeted toward general security specialists and not specifically intended for use by software developers. In addition, this approach does not provide any direction or mechanisms that support its integration with software engineering practices outside of the security domain. Finally, there is no specification or tool for rigorous, traceable transition from model to code.

3) Extensions to UML.: There are a number of approaches to modeling security protocols that extend UML in order to create profiles specific to the security domain. These UML extensions take into consideration the concern of how to effectively address integrating security as a component of the overall system design in order to reduce the risk of introducing vulnerabilities during implementation. UMLsec is an approach that formally specifies security requirements through stereotypes, tagged values and constraints to model security requirements such as secure links, data security, critical and fair exchange, to name only a few [3]. SecureUML is an extension for specifying access control policies for actions on protected resources through the use of a security modeling language where the abstract syntax and semantics allow it to be combined with design modeling languages other than UML [6]. The UML extension for security protocol (USP) is a framework that encapsulates knowledge for modeling of security protocols through two separate profiles, one representing physical organization and the other behaviours [7]. SecureMDD is a model-driven method for the development of secure applications utilizing UML extended by a UML profile and a programming language, Model Extension Language (MEL) [8].

The strengths of approaches that utilize and extend UML is their foundation in a common, widely used standard for object-oriented modeling that has been adopted within industry. However, UMLsec and USP do not provide a rigorous, traceable and automatic transition from design to code. Only SecureUML and SecureMDD, utilize the concept of MDD and the use of executable models. In the case of SecureUML, the approach does not use any specific tool set in the generation of code from design and the focus is on Role-Based Access Control (RBAC). Similarly, SecureMDD is an application specific approach for the integration of security into smart card systems, where the translation of design to code currently only generates Java Card code.

The body of research examining the use of UML for modeling security protocols has focused on extending UML to provide direct support for the security domain. Given that several different variations on extensions to UML do exist, no single one has emerged as a de-facto standard. Many of the approaches described above are based upon older versions of UML and those that do support newer versions, including MDD, are specific to a particular type of security application and not more broadly applicable.

III. UML 2 AND MDD FOR THE MODELING OF SECURITY PROTOCOLS

Our approach was to develop a modeling technique using UML 2, without extensions, taking advantage of the new
features within this version that support the definition of communication between participants through ports and protocols and provides the ability to create an executable model. The use of MDD to construct executable models provided a means to examine a mechanism to bridge the current gap that exists between the design and implementation. As part of the modeling exercise, we evaluated whether or not the security protocols modeled in UML 2 were expressive enough to present the key concepts of a security protocol without losing any important details. In addition, we evaluated the use of executable models in order to determine if they were predictive in that they would reveal any non-obvious properties of the security protocols being modeled.

A. The Security Protocol Framework

In developing our approach we needed to model multiple security protocols so that we would have a variety of protocols that could be reviewed and evaluated during this process. In order to support modeling multiple security protocols, we constructed a flexible framework that could be used to model different security protocols. Included in the development of the framework was the requirement to include a threat model that adheres to the Dolev-Yao threat model, where the adversary is an active saboteur in that they may impersonate another principal, initiate communications or alter or replay a message that is being sent [9]. The assumptions made in constructing the framework were as follows:

- It is assumed that for all encrypted messages, no principal will be able to obtain the plain text or create encrypted text without having the correct private key;
- All principals, Alice, Bob and the Adversary, can be legitimate participants but in stating that, we cannot assume that all legitimate participants will act honestly and follow the protocol specification;
- The Server is a trusted participant in the environment and will execute the protocol honestly and not engage in any activity that will deliberately compromise any other participant;
- The Server generates keys in such a way that it is sufficiently random so that a participant, who does not have access to that private key, cannot guess the key given that it is a random number that has been chosen from a sufficiently large space; and
- The Adversary is not just a passive eavesdropper but is also an active saboteur and follows the Dolev-Yao threat model as described by Mao [10].

The modeling tool that we elected to use to construct our framework was IBM’s Rational Rose RealTime®. The Rose-RT tool set supports the Real-Time Object-Oriented Modeling (ROOM) as first introduced by Selic et al [11]. Within the Rose-RT tool set, components are equated with capsules where capsules are Object Oriented (OO) classes that can only communicate through typed ports thereby limiting the communications between participants. Ports provide a means to describe a protocol class that defines the messages that are sent and received by capsules dependent on the direction of the port.

1) Design of the Security Protocol Framework: The design of the framework was partitioned based on the type of participant, where the Server was treated separately from other participants given that it is a “trusted” participant responsible for generating session keys. The remaining participants, *Alice, Bob* and the *Adversary*, shared common behaviours and so were treated as similar, with the idea that there would be a parent capsule, Participant, and the other participants would inherit the common behaviours from the parent. All capsules would be contained within the single capsule, SecureEnvironment, which would be the overall container used to automatically generate the code.

The capsule, Participant, contains the ports that allow communication between participants and the Server. The two main ports are ParticipantPort and ServerPort, which are associated to two protocol roles: ParticipantProtocol and ServerProtocol, where protocol roles specify the set of messages that can be received or sent from a port. A port plays the role of one participant in the communication relationship, so a port has an associated direction which will either be base/client or conjugate/server. Only compatible ports can be connected such that every signal in the ‘in’ set of one protocol role must be in the ‘in’ set of the other protocol role.

The challenge in designing the framework was to be able to model the Adversary correctly in order to adhere to the Dolev-Yao threat model. The Adversary is a participant much like *Alice* and *Bob* and shares common behaviours, but there is an additional requirement unique to the Adversary and that is its ability to eavesdrop on all messages sent between participants, including those to and from the Server. This constraint influenced the design of the framework in order to ensure the Adversary could function as a “middle man” in all communications.

2) The Adversary: The Adversary controls the communications between all principals and can therefore obtain any message that is passing between principals. This requires that all communications between *Alice, Bob* and the Server must be accessible to the Adversary. In order to model this behaviour within the framework, the Adversary had to have access to the messages that are sent via the ports for each of the capsules. The design decision was made to embed the capsules for *Alice* and *Bob* within the Adversary capsule through an aggregate relationship. The Adversary contains internal ports through which *Alice* and *Bob* can communicate with one another or communicate with the Server. The intent of this design is that the Adversary becomes synonymous with the network, in that all communications pass through the Adversary and are routed accordingly. In Fig. 1, the capsules and relationships are described.

3) Communications: Ports and Protocols: Communications between all participants occurs via ports representing a protocol class. In the framework, these ports are not directly connected or wired, requiring that a participant must first establish a connection to a port and then disconnect when the port is no
longer required. All capsules that inherit from the Participant
capsule have a port for communicating with the Server and
one for communicating with another participant. Given that
the Adversary contains PrincipalA and PrincipalB, the Adver-
sary capsule contains two internal ports that PrincipalA and
PrincipalB connect to when establishing communications with
another participant or the Server. The Adversary has internal
mechanisms to track these communications in order to ensure
that messages are directed to the correct participant in the
exchange. These internal mechanisms also provide the means
for the Adversary to either eavesdrop or launch an attack as
may be appropriate to the security protocol scenario being
modeled.

There are two protocols used for communications within
The ServerProtocol is a binary protocol and a participant, using
the services of the Server, must have a conjugate port that
corresponds to the Server’s public base port. Given the limited
role that the Server plays within the security protocols being
modeled, the protocol contains one ‘in’ signal, RequestKey,
and one ‘out’ signal, SendKey. The ParticipantProtocol is
different from the ServerProtocol in that it is a symmetric
protocol, where the set of ‘in’ messages maps directly to
the set of ‘out’ messages. This means that participants com-
municating with other participants through a port with the
ParticipantProtocol role communicate through base ports, no
conjugate port is required to define the flow of messages. This
allows participants to both send and receive messages through
their ports.

4) State Machines.: To model the behaviour of the par-
ticipants within the protocol each participant is defined by a
Behavioural state machine that describes reactions to events
and resulting behaviours. The state machines vary based upon
the participant in the protocol. The Server is represented by a
very simple state machine containing a single state with a self-
transition that occurs when it receives a serverRequest signal.
In contrast, all other participants inherit the Behavioural state
machine of the Participant capsule, which contains only two
states: Idle and Communicating.

It is the state machines that are defined for the participants,
PrincipalA and PrincipalB, that are critical in terms of mod-
eling the security protocols. While PrincipalA and PrincipalB
inherit the high-level state machine from the Participant cap-
sule, it is the Communicating state that is further specialized
into those states that represent the steps within the security
protocol. For each security protocol, it is this hierarchical state
machine that must implement the concepts of UML 2 Protocol
state machines but in the form of a Behavioural state machine.
The UML 2 Protocol state machines model the ordering of
operation calls on an object, without describing associated
behaviours. The approach used in this framework is to adhere
to the concept of the Protocol state machine by specializing it
using a Behavioural state machine, allowing both the sequence
of messages and associated behaviours to be described.

5) Modeling of Threats.: The Adversary in the framework
is one of the most intricate components of the model due to
its omniscient role within the security protocols. It contains
the ability to execute various threats during protocol runs in
order to determine if one of these threats can compromise
the security protocol in any way. In order to support multiple
threat models the Adversary has a hierarchical state machine
that decomposes the Communicating state into two possible
states: Eavesdropping or Attack, as described in Fig. 2.

When in the Eavesdropping state the Adversary simply acts as the “middle man” in relaying messages between participants and tracks those messages as part of the protocol run. In order to launch an attack, the Adversary must be in the Attack state, which is further decomposed into a state machine containing a series of decision points or control points where it is determined which threat to execute, as illustrated by Fig. 3. Note that each attack state is also a composite state in that it contains states and/or self-transitions that are specific to that type of attack.

The Adversary’s attack state can contain multiple types of attacks but only one attack will be executed during a security protocol run. The state machines for each threat type only redefine those transitions that are applicable to that particular threat. If a transition is not redefined within the threat state, then Rose-RT will look for an inner internal self-transition, as indicated by the dashed line, on the enclosing superstates. The execution of one of these inner internal self-transitions does not cause the state machine to exit or re-enter the state in which it is defined.

In order to be able to control the types of threats that can be launched, the Adversary has an internal capsule named ThreatController that maintains a list of potential threats that can be executed on each subsequent run of the security protocol. At the end of each run, the Adversary checks the ThreatController to determine if it should move into an attack state or remain in the Eavesdropping state. The use of the ThreatController is important as not all threats are applicable to every protocol, so when executing a security protocol it may not make sense to apply all possible attacks.

The attacks that were modeled within the framework were limited to those that applied to the security protocols being modeled as part of the construction and evaluation of the framework. The rationale for limiting the modeling of threats was that in evaluating a protocol it should be validated against those goals that the designer intended the protocol to achieve. Applying a set of threats outside of the scope of the protocol goals may yield interesting results, but in this research the threats were kept to a finite set in order to focus on evaluating the framework and the use of UML 2 and MDD to model security protocols.

The framework can accommodate multiple attack types in order to expand the set of available threats, since an attack is simply a state within the overarching Attack state. Adding a threat simply requires the addition of a choice point for that type of attack, the addition of the attack name in the ThreatController attack list and then the definition of the specific attack state. The attacks that were modeled for the framework included the following:

- **Eavesdropping** - This is the most basic attack and by default the Adversary is in this passive attack state when there is no other threat being executed. In this type of attack, the Adversary can capture the information relating to the message flows between principals and any information in those messages that is not encrypted.
- **Intercept** - An Intercept attack involves intercepting a message from one principal to another, excluding the Server, and interjecting the Adversary’s identity into the message resulting in confusion as to who is involved in the protocol and who knows the session key.
- **Alteration** - In this attack, the Adversary intercepts a request to the Server and inserts its own identity. The result is the Server issues a session key that exists between the Adversary and the other principal involved in the protocol, allowing the Adversary to obtain a copy of the session key.
- **ServerReplay** - If we assume that the Adversary can obtain the value of a session key used in a previous protocol run, then replaying an old session key would allow the Adversary to intercept and decrypt any messages sent between the principals once the security protocol has
established communications.
- **Interleave** - The interleave attack is a threat that is specific to a flaw that was discovered in the Needham-Schroeder protocol as described by Lowe [12]. The idea is that the *Adversary* initiates a legitimate security protocol run with PrincipalA and then in turn the *Adversary* initiates a fake run with PrincipalB masquerading as PrincipalA and using information obtained from the legitimate protocol run with PrincipalA.

6) **Code Generation.** For all of the security protocol models defined we used the Rose-RT tool to generate the code from the models. The only code inserted into our protocol models was specific to the generation and handling of the messages. For PrincipalA and PrincipalB we had to create code to manage the parsing, construction and validation of the messages as per the security protocol specifications. No code was manually created to support the transition between states for any of the participants, this code was all generated through the Rose-RT tool.

B. **Modeling Security Protocols**

A series of simple security protocols were modeled in order to develop the framework and ensure flexibility in the design. The simple protocols all contained known flaws that were used as a basis to construct the threats that are contained within the *Adversary*. The development of the framework was incremental and corresponded with each successive security protocol becoming more complex in its interactions. With a basic framework in place, the two protocols: Needham Schroeder and Yahalom were modeled and executed. The Needham-Schroeder protocol was selected as it contains a known and documented flaw as described by Lowe [12]. Whereas the Yahalom protocol was selected as a security protocol that did not have any documented flaws. We will examine the implementation and execution of the Needham-Schroeder protocol given it’s well documented vulnerability in order to demonstrate how our security protocol framework can be applied and the resulting models executed.

1) **Modeling the Needham-Schroeder Protocol.** In the Needham-Schroeder security protocol, nonces are used to ensure freshness in order to guard against a Replay attack. The protocol run in standard notation appears as follows:

1) $A \rightarrow S : A, B$
2) $S \rightarrow A : \{K_{PB}, B\}K_{AS}$
3) $A \rightarrow B : \{N_{A}, A\}K_{PB}$
4) $B \rightarrow S : B, A$
5) $S \rightarrow B : \{K_{PA}, A\}K_{BS}$
6) $B \rightarrow A : \{N_{A}, N_{B}\}K_{PA}$
7) $A \rightarrow B : \{N_{B}\}K_{PB}$

Within the framework it is the state machines for PrincipalA and PrincipalB that reflect the steps within the Needham-Schroeder protocol. The Behavioural state machines reflect the Protocol state machines in that the state transitions only occur based upon receiving the correct message in the correct sequence. As an example, PrincipalA initiates the protocol run by requesting the session key from the *Server* as in step 1 of the protocol run described above. It only transitions to the state, sendMessageToB, where it formulates and sends the message to PrincipalB, once it has received the correct response from the *Server* as in step 2. Figure 4 illustrates the Behavioural state machine of PrincipalA as designed for the Needham-Schroeder security protocol. PrincipalB’s Behavioural state machine corresponds to that of PrincipalA in that PrincipalB only transitions to the next state when it receives the correct message from PrincipalA or the *Server.*

---

**Fig. 3. Attack State Machine**

The diagram illustrates the attack state machine with various attack types such as `Intercept`, `startAttack`, `endAttack`, `interceptAttack`, `startAttack`, `True`, `alteractionAttack`, `replayAttack`, `interleaveAttack`, etc. Each state transition is marked with the relevant attack type, indicating the sequence of events and the conditions under which they occur.
2) Applying Threats to the Needham-Schroeder Protocol.: We applied all of the threats to the Needham-Schroeder protocol, but prior to executing the model we predicted the outcome based upon the attack type and the structure of the protocol. The only difference was in the way in which we implemented the InterLeave attack as we designed it according to Lowe’s description of the attack on the Needham-Schroeder security protocol [12]. In this attack Lowe assumes that all participants know the other principals’ public key, so we can reduce the set of messages in the protocol to the following:

1) \( A \rightarrow B : \{N_A, A\} K_{PB} \)
2) \( B \rightarrow A : \{N_A, N_B\} K_{PA} \)
3) \( A \rightarrow B : \{N_B\} K_{PB} \)

So we had two versions of the Needham-Schroeder protocol, where we determined the threats that applied and predicted the outcomes as described in Table I.

C. Validation

Once the framework was constructed, we used the Needham-Schroeder Public Authentication protocol to determine if the expected behaviours and known flaws were revealed within the model. We also modeled the Yahalom protocol, a security protocol with no known flaws, in order to evaluate if the model was expressive and predictive. Initially, we applied an informal visual inspection approach to the model. Then we executed each of the models and from the execution examined the state monitor, sequence diagrams and utilized the trace capture in order to examine the artefacts to verify if the expected behaviours of the protocol are evident during execution. For all security protocol models created, we subjected them to adversarial behaviours with the intent that this would reveal any deficiencies in the design or the actual model itself. By comparing what the visual model reveals about the security protocol with actual behaviours exposed during execution, we evaluated if there are any deficiencies with our approach of using UML 2, with no extensions, and MDD for the modeling of security protocols.

D. Findings

We were able to use UML 2 to model the simple protocols and the Needham-Schroeder and Yahalom protocols without requiring any extensions to the language. The approach of building the framework, using progressively more complex protocols, demonstrated that UML 2 could support all of the protocols modeled without revealing any circumstances under which the language was deficient in supporting modeling within this domain.

1) Informal Visual Inspection.: The validation of the security protocols through an informal visual inspection process revealed that, while UML 2 could support modeling within this domain, the visual models did not provide any more information about the protocol than was revealed through the definition of the security protocols using the standard notation. In fact, the UML 2 visual models created using Rose-RT revealed less information about the security protocol given that the state machine transitions were identified only with a user defined label, which did not reveal anything about the underlying signal/message or its contents.

2) Execution of Models: In terms of analyzing and validating the security protocols, the ability to execute the model provided the greatest ability to reveal non-obvious properties of the protocol. The Rose-RT tool provides a facility to capture the sequence diagram relating to the protocol execution and this diagram can be analyzed to examine the protocol run under normal conditions and when the threats are applied. The sequence diagrams produced logged the flow of the messages and the change of states for the various participants involved in the protocol run.
TABLE I
NEEDHAM-SCHROEDER PROTOCOL: THREATS AND EXPECTED OUTCOMES

<table>
<thead>
<tr>
<th>Attack Type</th>
<th>Needham-Schroeder Protocol</th>
<th>Needham-Schroeder Abbreviated Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>Unsuccessful</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>Alteration</td>
<td>Unsuccessful</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>ServerReplay</td>
<td>Successful</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>InterLeave</td>
<td>Not Applicable</td>
<td>Successful</td>
</tr>
</tbody>
</table>

TABLE II
SECURITY PROTOCOLS AND THREATS

<table>
<thead>
<tr>
<th>Attack Type</th>
<th>Needham-Schroeder Protocol</th>
<th>Needham-Schroeder Abbreviated Protocol</th>
<th>Yahalom Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>Attack</td>
<td>Unsuccessful</td>
<td>Attack</td>
</tr>
<tr>
<td>Alteration</td>
<td>Attack</td>
<td>Unsuccessful</td>
<td>Attack</td>
</tr>
<tr>
<td>ServerReplay</td>
<td>Attack</td>
<td>Unsuccessful</td>
<td>Attack</td>
</tr>
<tr>
<td>InterLeave</td>
<td>Successful</td>
<td>Not Applicable</td>
<td></td>
</tr>
</tbody>
</table>

By applying a series of threats over multiple protocol runs, we were able to compare our predictions about expected outcomes with the resulting sequence diagrams. This supported our ability to determine if the implementation adhered to our protocol specification and whether any unexpected behaviours were revealed. The advantage of having a set of potential threats available to apply to the model, is that we can then test the model under various conditions and observe the behaviours of the implementation of that security protocol. In our research, we developed a limited set of threats but this provided a basis for the testing of our models under adverse conditions even when the threat was not expected to be effective against a particular protocol.

Table II illustrates the set of threats available within this version of the framework and the vulnerabilities of those security protocols that were implemented. Note that in the case of the Needham-Schroeder abbreviated protocol we only applied two of the threat types: Eavesdropping and the InterLeave attack. For the Yahalom protocol, the only threat not executed was the Interleave attack given that the protocol under attack relied on public keys as the encryption mechanism.

IV. CONCLUSION
We were able to use UML 2 to model the simple protocols and the Needham-Schroeder and Yahalom protocols without requiring any extensions to the language. The approach of building the framework, using progressively more complex protocols, demonstrated that UML 2 could support all of the protocols modeled without revealing any circumstances under which the language was deficient in supporting modeling within this domain.

ACKNOWLEDGMENT
This research was supported in part by the Department of National Defence, Technical Airworthiness and Engineering Support.

REFERENCES