A Model-based Approach to Security Flaw Detection of Network Protocol Implementations

Yating Hsu, Guoqiang Shu and David Lee
Department of Computer Science and Engineering, The Ohio State University
Columbus, OH 43210, USA
{hsuya,shug,lee}@cse.ohio-state.edu

***Abstract*— A lot of efforts have been devoted to the analysis of network protocol specification for reliability and security properties using formal techniques. However, faults can also be introduced during system implementation; it is indispensable to detect protocol implementation flaws, yet due to the black-box nature of protocol implementation and the unavailability of protocol specification most of the approaches resort to random or manual testing. In this paper we propose a model-based approach for security flaw detection of protocol implementation with a high fault coverage, measurability, and automation. Our approach first synthesizes an abstract behavioral model from a protocol implementation and then uses it to guide the testing process for detecting security and reliability flaws. For protocol specification synthesis we reduce the problem a trace minimization with a Finite State Machine model and an efficient algorithm is presented for state space reduction. Our method is implemented and applied to real network protocols. Guided by the synthesized model our testing tool reveals a number of unknown reliability and security issues by automatically crashing the implementations of the Microsoft MSN instant messaging (MSNIM) protocol. Analytical comparison between our model-based and prevalent syntax-based flaw detection schemes is also provided with the support of experimental results.**

***Index Terms*— formal model, fuzz testing, protocol implementation, and security flaw**

I. MOTIVATION

With the reliability and security as the main goal of network protocol design and implementation methods and tools for detecting such flaws are urgently needed [13] [24]. According to US-CERT [25] in 2006 alone a total of 600,000 messages reported security incidents. Among the 2,453 computer vulnerabilities in the database, around 1,000 are related to network protocols. CSI/FBI also reported [11] that among 597 companies surveyed 66% of them are currently using penetration testing techniques and automated tools with the hope of finding security flaws early.

While the extensively studied protocol validation techniques [8] help to discover problems in the design, flaws may also be introduced to the implementation during the phase of coding, deployment and integration. A major cause of flaws in protocol implementation is improper handling (e.g. incorrect assumption) of input data. The resulting security exploits range from the well understood buffer overflow attack to more sophisticated protocol level attacks such as enforcing a lower (unsafe) version of handshaking protocol during version negotiation and soliciting confidential content using illegitimate queries. Consequently, we want to construct input sequences that either trigger its insecure interactions with its peers or bring it to unreliable states. However, the prevalent security testing for protocol implementation faults is rather limited due to the following two major hurdles:

- **Black-box Implementation** The black-box nature of protocol implementation poses a challenge. Software security testing methods [9] [10] often look for suspicious operations in source code and derive input data to reach them. However, for protocol implementations usually only very limited knowledge is available especially at post deployment and integration stage.
- **Lack of Formal Specification** A complete and machine understandable protocol specification is very rarely available, which makes formal protocol testing [16] [17] infeasible. Even when a specification is manually derived there is no guarantee that the corresponding implementation conforms to the specification since it might contain unspecified engineering details [24] or discrepancy from system evolvement.

To cope with the difficulties a variety of techniques have been developed. For instance, fuzz testing works by mutating a portion of the normal input data at the ingress interface of a protocol component in order to reveal unwanted behaviors [19]. It becomes increasingly popular in recent years as a means of security flaw detection due to extremely low cost and proven effectiveness. However, existing black-box protocol fuzzing tools are designed in a protocol specific and ad-hoc manner. The selection of input messages from a session to mutate is done either *randomly* or *manually*. The unstructured selection of input makes it impossible to measure the comprehensiveness (i.e. how much is covered) of a test suite, and relying on human interfaces usually makes such testing tools hard to automate.

We propose a formal and automated method. In a preliminary work [23], we showed the feasibility of our approach. In this paper we investigate a general theory and algorithms for automated protocol implementation synthesis and fuzz testing for reliability and security flaw detection and report experimental results on real and popular protocol systems.

1-4244-2507-5/08/$20.00 ©2008 IEEE

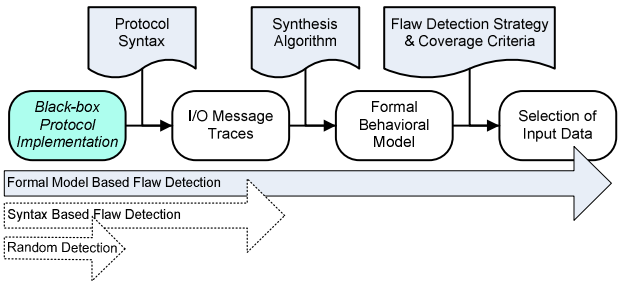

Fig. 1. Workflow of Security Flaw Detection

II. OUR APPROACH

We propose a testing method for improving the quality and measurability of security flaw discovery based on an automatically synthesized formal protocol specification. The key idea is to first obtain an approximate behavior model of a protocol implementation under test and then use it to guide input selection for fault coverage. Fig. 1 illustrates the workflow of this two-phase approach. After network traces are represented by abstract input/output message symbols, a formal behavioral model is synthesized for capturing the design aspects of the protocol. The model can be constructed using either active black-box checking technique on protocol implementation or passive monitoring of input/output traces. We study the tradeoff of the two approaches and adopt a passive synthesis algorithm based on trace minimization. The synthesized formal model represents in a succinct way the key states and transitions of a protocol implementation that systematically guides the flaw detection process.

Fault coverage criteria based on a behavioral model has fundamental advantages over criteria from protocol message syntax. Although the latter is easy to implement, it is in general inaccurate. Messages of a same type may serve quite different roles in a protocol session. For instance, an ACK packet in TCP can trigger different processing logics depending on the receiver's current state of congestion control mechanism. Consequently, the same input serves different roles when fault discovery is concerned and, therefore, should be distinguished in the test selection.

We implement the proposed methodology and build a prototype of a testing tool that is different than the prevalent ones. To our best knowledge, this is the first automated tool providing measurable coverage and universal applicable to all network protocols. Extensive experiments are conducted on real implementations, including the popular MSN instant messaging protocol (MSNIM) client site. Our tool automatically constructs a number of input sequences, which cause the implementations to crash under a set of input fuzzing functions. A quantitative comparison of the coverage using the synthesized model and the syntactical message type is presented to justify our argument with supporting experimental results.

The rest of this paper is organized as follows. Section III contains the formal definitions of a protocol model and the flaw detection problem, followed by a discussion of our assumptions. Section IV focuses on the specification synthesis approaches. We first briefly review an active learning algorithm and discuss its limitations. We then present a four-step passive synthesis method based on reduction of tree FSM model. We describe in detail a polynomial time algorithm. Section IV defines coverage metrics with the FSM model using fuzz testing as a case study. Section V reports the experimental results on fuzzing MSNIM protocol implementations. Related works are discussed in Section VI. The final Section VII concludes this paper with a discussion of future works.

III. A FORMAL MODEL

After defining a formal model, we discuss flaw detection process.

A. Protocol Implementation and Abstract Model

A protocol is viewed as a set of components interacting with each other through message exchanges. In this work we focus on testing black-box implementation of a single component, often called deliverable testing. A protocol has its message syntax. Denote all valid input and output messages (byte strings) as MSG_I and MSG_O, respectively. Thus an implementation B is defined by a computable function f_B: $MSG_I^* \rightarrow MSG_O^*$ where B starts from its initial state.

In order to represent a synthesized model, we first discuss message abstraction since the raw message set often has a formidable size. Let A_I and A_O be finite sets of input and output symbols, respectively. Denote two abstraction functions $\alpha: MSG_I \rightarrow A_I$ and $\beta: MSG_O \rightarrow A_O$, which map the messages to symbols. Functions α and β can be trivially generalized and defined on message sequences. We say that an abstraction α and β is deterministic w.r.t. an implementation B *if and only if* for any $x,y \in MSG_I^*$, $\alpha(x) = \alpha(y) \leftrightarrow \beta(f_B(x))=\beta(f_B(y))$. That is, I/O mapping and abstraction are commutative.

Given A_I and A_O, an approximate abstract model of B is a Finite State Machine (FSM) $<S, s_0, A_I, A_O, f_{next}, f_{output}>$, where S and s_0 are state set and initial state, A_I and A_O are input and output alphabet, $f_{next} : S \times A_I \rightarrow S$ is the transition function and $f_{output} : S \times A_I \rightarrow A_O$ is the output function [17]. Both f_{next} and f_{output} can be partially defined. Function f_{outout} is generalized in a straightforward way for calculating an output sequence, given an input sequence from s_0. We call an FSM a tree FSM if its state transition graph is a tree. A trace produced by an FSM is a sequence of input/output message pairs, i.e. $tr = \{<x_1,y_1>, <x_2,y_2>, \ldots\}$, $x_k \in A_I$ and $y_k \in A_O$.

B. Flaw Detection Problem

Our flaw detection process involves three parts: (1) A black-box protocol implementation B; (2) A predefined predicate $GOAL: MSG_O^* \rightarrow \{true, false\}$ that indicates which input sequence to B from its initial state results in a flaw, represented by *true*, i.e. insecure or unreliable behavior; and (3) An input sequence selection or construction strategy that reduces the search space to a subset $\Phi \subset MSG_I^*$.

The procedure of model-based flaw detection is to first construct an FSM model from a black-box implementation B, and then use it to facilitate searching for a message sequence

$seq \in \Phi$ such that $GOAL(f_B(seq))=true$. Even with restrictions testing on all sequences in Φ is usually not realistic due to its formidable size. The challenge is how to intelligently and systematically select a small number of representative input sequences from Φ, taking advantage of the constructed formal model. We apply protocol fuzzing where the heuristic subset is defined to include all the sequences that can be transformed from an observed trace under a message fuzzing function Z: $MSG_I^* \rightarrow MSG_I^*$.

We note that the *GOAL* predicate can be defined in different ways depending on specific security properties. For instance, to test for simple observable failure, such as crashing, the predicate checks whether a special output symbol of protocol system crash is found. However, for message confidentiality testing [24], the predicate must calculate the attacker's knowledge with all output messages.

C. Assumptions

In this section we describe our assumptions on protocol implementations and fuzzing. First, protocol implementations are deterministic, that is, the I/O mapping f_B from the initial state is a function. This assumption holds for most of the protocol implementations and enables us to eventually synthesize a deterministic FSM model. Second, since in our model input and output message abstraction is done by decoding/encoding of a subset of message field, we assume that this can be done automatically. That is, a protocol message parser is available, either derived from the syntax manually or provided by other protocol engineering tools. In practice, tools such as Ethereal is able to parse a variety of popular protocols

Another assumption of fuzz testing experiments is on how the input messages are modified. Abstract input symbols in A_I are formed by removing some session related fields from a message in MSG_I. The reverse transformation is in general infeasible. That is, during testing we cannot construct arbitrary input messages even when its corresponding input symbol is known. Our approach for forming a new valid message is to apply a fuzzing function to a monitored message.

Finally, we assume that the execution of implementation B can be fully controlled (automatic start/reset and termination) that enables us to repeat the flaw detection experiment with different settings systematically.

IV. PROTOCOL SPECIFICATION SYNTHESIS

This section contains our main technical approach. As indicated, a key to achieve measurable coverage is to synthesize an *abstract* and *approximate* FSM model M_x of B that is used to guide the test selection process. M_x should ideally be an abstraction of all the observed behavior of B, and its alphabets A_I and A_O are formed by removing protocol message field details. We first briefly review an active synthesis algorithm using machine learning and discuss its limitations. Then we focus on a passive synthesis algorithm.

A. Review of an Active Learning Approach

The problem of protocol synthesis from I/O behavior is coupled naturally with machine learning. In our earlier work [24][23], we studied a supervised learning algorithm following the theoretical insights of [2][21]. An estimation model M_x^* of the implementation B is initialized as a single-stated FSM. M_x^* is updated as more traces are discovered according to the supervised FSM learning algorithm L^*_{fsm} (a modification of Angluin's classic L^* algorithm). A conformance test generator serves the role of a teacher that provides traces as counter-example – showing the difference between M_x^* and B. The counter-example is then used to prepare for the next estimation that is supposedly more accurate, that is, closer to the behavior of B. This iterative process terminates when no more counter-examples can be found for continued learning.

Due to the elegance of L^* algorithm this learning approach has very attractive properties: the estimation M_x^* is always promising in the sense that if the abstraction of B is indeed equivalent to an FSM of N states and P input symbols, at most $(N+P)$ estimations will be made before the algorithm terminates.

On the other hand, this approach is not practical for many real problems due to the following two reasons. First, because L^* (and hence L^*_{fsm}) algorithm requires active input query to B, the tester needs to construct actual message (in MSG_I) from the abstract symbol (in A_I). In practice it is not always possible due to the difficulty of padding missing fields with meaningful values. Second, the success of learning relies on the idealization of teacher, i.e. the teacher can find counter-example efficiently. However the cost of teacher's algorithm is high and in the worst case it may fail to find counter-examples [24], causing the learning process to terminate with an inaccurate model.

B. Passive Synthesis with Partial FSM Reduction

In the rest of this section we focus on a synthesis method that is fundamentally different than the learning-based approach. The new method only requires *passive* monitoring of B with the anticipated tradeoff that the quality of synthesize model depends on the monitored traces.

Given a set of traces, ideally we want to find a minimized FSM that contains all the traces and only these traces. However, the problem is NP-hard [17].

We present an algorithm that constructs in polynomial time a reduced FSM – not necessarily minimized – that contains the given traces and only these traces. This process contains four steps: (1) A large number of traces are gathered and preprocessed by removing the session dependent fields; (2) Each trace is pre-processed for identifying possible loops; (3) Construct a tree FSM M_T that accepts all the resulting traces and only these traces; and (4) Merge equivalent states in the tree FSM and obtain a reduced and equivalent FSM M_X . M_X is taken as a final model for deriving test sequences for flaw detection.

Step (1) The execution of target protocol implementation B is monitored for a period of time and all input/output traces are recorded. Each trace is a sequence of message pairs $<x_i,y_i>$, where $x_i \in MSG_I$ and $y_i \in MSG_O$. The traces are likely to be all distinct; however some of them are essentially the same

except for the session related fields (timestamp, nonce, etc.). These fields should be abstracted from the messages. Moreover, the tester might manually remove additional fields from the trace so long as the abstractions are deterministic.

Step (2) Since the lengths of the monitored traces are finite, it is impossible to exactly identify loops. We make the following assumption on monitored loops. Given a trace, any repetitive and consecutive sub-traces are assumed to be from a loop. For instance, in a trace *uababcv*, we assume that sub-trace *ab* is from a loop and the trace is: $s_0 u s_1 a s_2 b s_1 a s_2 b s_1 c s_3 v s_4$ where s_i, $i=0, 1,2,3,4$, are states and a loop of length *2* is observed that is repeated twice from state s_1: $s_1 a s_2 b s_1 a s_2 b s_1$. The repeated sub-trace (substring) identification problem was well studied in string matching literature and efficient polynomial time algorithms were developed [6]. We process each trace and remove all the loops but record the location and content of the loops. After Step (4), we restore all the loops and obtain an FSM model for flaw detection.

Step (3) Given a loop-free set of traces from Step (2), we construct a tree FSM that accepts all the trace and only these traces as follow. Starting from an empty FSM with only an initial state s_0, traces are added one after another. Assume that α and β are specified as the initial abstraction of MSG_I and MSG_O. The algorithm is given below.

Algorithm 1 (Construction of Tree FSM)

Input: implementation *B* and initial abstraction α and β;
Output: a tree FSM M_T;
begin
1. $M_T = <\{s_0\}, s_0, \alpha(MSG_I), \beta(MSG_O), f_{next}, f_{output}>$;
2. **for each** monitored trace $\{<x_i, y_i> | 1 \le i \le L\}$ **do**
3. $s=s_0$;
4. **for** i=1 **to** L **do**
5. **if** ($f_{next}(s, \alpha(x_i))$ is not defined)
6. add a new state s_{new} to M_T;
7. add a new transition $(s, s_{new}, \alpha(x_i), \beta(y_i))$ to M_T;
8. s=s_{new};
9. **else if** ($f_{outupt}(s, \alpha(x_i)) \neq \beta(y_i)$)
10. refine α and/or β;
11. restart the algorithm from line 1;
12. **else** $s= f_{next}(s, \alpha(x_i))$;
end

For each trace (line 2), Algorithm 1 follows a prefix of it that is already in the tree FSM (line 12). If an input message is not defined, a new branch is created (line 5-8). If the current abstraction of messages appears to be non-deterministic, the discrepancy of output will be detected (line 9). In this case we have to refine α and/or β by adding more fields in abstract symbols to make it deterministic. With a finite number of traces this algorithm terminates with a tree FSM with a cost proportional to the total length of traces. Obviously, each trace is accepted by the FSM and each path from the root represents a trace or prefix of a trace.

Step (4) Given a large number of monitored traces, the resulting tree FSM can be big and we want to reduce its size. It is NP-hard to minimize an arbitrary partially specified FSM [17]. It is also NP-hard to minimize an arbitrary partially specified tree FSM for the following reasons. A tree FSM is equivalent to and is a different representation of a given set of traces. Therefore, finding a minimized machine of the tree FSM is equivalent to finding a minimized machine that accepts all the given traces and only these traces. Therefore, it is NP-hard to minimize partially specified tree FSM.

We now present an algorithm that merge equivalent states in a tree FSM M_T to reduce its size. Note that for partially specified machine merging equivalent states alone may not result in a minimized machine [17].

Given a state (node) in a tree FSM, an input sequence is valid if it takes the machine along a tree path from that state. Two states are equivalent if for any valid input sequence the two states produce the same output sequence. Consequently, two states are equivalent if and only if the two subtrees rooted at the two states are isomorphic. As a special case, two subtrees of height 1 are isomorphic if there is a one-to-one correspondence between their tree edges and the corresponding edges have the same I/O symbols. A state is of height 1 if the subtree rooted at the state is of height 1.

We now identify all the equivalent states of a given tree FSM bottom up as follows. We first examine all the states of height 1 and identify equivalent states by a pair-wise comparison. Then we color the states of height 1 such that two states are equivalent if and only if they have a same color. We then shrink all the processed subtrees of height 1 into a node (of height 0) with the color of its root node. We now have a tree FSM of height decreased by 1. However, the leaf nodes represent subtrees now. Consequently, two subtrees of height 1 are isomorphic if there is a one-to-one correspondence between their tree edges and the corresponding edges have the same I/O symbols; furthermore, their corresponding incidental leaf nodes have a same color. We repeat the process of identifying equivalent states of height 1 bottom up until we obtain a tree of height 1.

We now expand and recover the tree FSM and each node maintains the color it has obtained during the process. We now examine all the nodes and their colors and merge nodes of a same color top down as follows. If two nodes u and v have a same color, we merge them as follows. We remove v and redirect all the tree edges, which end at v, to u. Since we merge nodes top down no nodes are involved in merging more than once. However, when the process terminates we obtain a DAG (Directed Acyclic Graph) FSM M_X. It is equivalent to the original FSM M_T but of reduced size. We summarize the procedure in Algorithm 2.

Suppose that the given tree FSM has n leaf nodes (states). A naive pair-wise comparison of subtrees of height 1 costs $O(n)$ and there are of order n^2 comparisons at each level, and the cost for processing at each level is $O(n^3)$. Instead, for each node of height 1, we assign a string to it by the color of each of its child node along with the I/O symbols on the tree edge to that child node. Since both the I/O symbols and colors have a range of order n, we can radix sort the associated strings with all the nodes of height 1 and identify equivalent nodes

(states) whose associated strings are identical. The cost is O(*n*). Therefore, for each level of processing in Line 2, the cost is proportional to the number of leaf nodes, and hence:

Proposition 1. The total cost of the reduction Algorithm 2 is proportional to the number of nodes of a tree FSM.

Algorithm 2 (Tree FSM Reduction)

Input: a tree FSM M_T;

Output: an equivalent and reduced DAG FSM M_X;

begin

1. **repeat**
2. **for** each pair of states *u* and *v* of height 1 **do**
3. **if** $u \equiv v$ /* equivalent */
4. assign a same color to *u* and *v*;
5. shrink all subtrees of height 1 to a leaf node;
6. **until** tree FSM is of height 1
7. recover tree FSM with nodes color reserved;
8. merge nodes of a same color top down and obtain
9. a DAG FSM M_X

end

We illustrate algorithm 2 using an example. Given four traces: tr_1={a,b,d,a}, tr_2={c,c,a,b,d,e}, tr_3={e,a,b,a,b,a,b,d,e} and tr_4={e,d,a}, we first identify loops. In tr_2 there is a self loop with {c,c} and in tr_3 a loop of length 2 with {a,b,a,b,a,b}. The loops are removed for the time being and we obtain: tr_2={a,b,d,e} and tr_3={e,d,e}. Applying Algorithm 1, we construct a tree FSM as in Fig 2(a). We initially color all the nodes white. We identify equivalent states of height 1 bottom up and have $s_4 \equiv s_5$; both are colored red. We shrink the two corresponding subtrees to leaf nodes. We repeat the process and have $s_3 \equiv s_2$; both are colored green. We now process top down and merge the two subtrees rooted at s_3 and s_2 with green color. We obtain a DAG tree in Fig. 2(b). Finally, we attach the loops and obtain a reduced machine M_X as in Fig. 2(c).

C. Coverage Criteria

Once we have synthesized a protocol specification M_X, it is used to guide input selection for flaw detection experiments. Again we use fuzzing as an example. Given a fuzzing function on input sequences, *Z*: $MSG_I^* \rightarrow MSG_I^*$, we apply it comprehensively to all steps in a protocol session.

Coverage for state-based models can be defined in various ways and we adopt one that is commonly used in practice for illustration and also for our testing tool. That is, a fuzzing function only mutates a single input message from a sequence (typically the last one). The mutated input corresponds to a transition in M_X. Given a test suite, *transition coverage* measures how many transitions are tested using this fuzzing function. Formally, given a set of input sequences of the following form whose last message is to be fuzzed: $Seq_k=I_k^1 I_k^2 \ldots I_k^{L(k)-1} I_k^{L(k)}$ where $I_k^j \in MSG_I$, and a synthesized model $M_X=<S, s_0, A_I=\alpha(MSG_I), A_O, f_{next}, f_{output}>$, the transition coverage is calculated as:

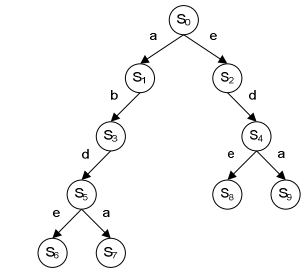

Fig. 2(a) Tree FSM after loop removal

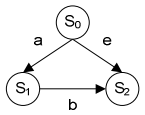

Fig. 2(b) DAG FSM without loops

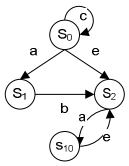

Fig. 2(c) Final result

$$TrCoverage(M_X < S, s_0, \alpha, f_{next} >, \{Seq_1, Seq_2, .., Seq_k\}) = \frac{|\{< s', I_k^{L(k)} > | s' = f_{next}(s_0, \alpha(I_k^1 I_k^1 \ldots I_k^{L(k)-1})) \wedge f_{next}(s', \alpha(I_k^{L(k)})) \downarrow\}|}{|\{< s, x > | s \in S \wedge x \in A_I \wedge f_{next}(s, x) \downarrow\}|}$$

The formula calculates the number of transitions in M_X, which are triggered by the last input symbol of an input sequence with a mutation, divided by the total number of transitions. The metric is in the range [0..1]. Transition coverage clearly depends on the quality of the selected sequences for covering more transitions. On the other hand, by the way M_X is constructed using monitored traces, obviously, a 100% transition coverage can be achieved if testing sequences are carefully selected.

V. EXPERIMENTAL RESULTS

We apply the proposed method and build a real protocol fuzzing tool. This section reports our experiments with the tool and its findings on one the most popular application protocols used in today's Internet: Microsoft MSN instant messaging (MSNIM) protocol.

A. MSN Instant Messaging Protocol

The MSNIM protocol (version MSNP9) is a proprietary text-based protocol developed and used for Microsoft MSN messaging services. In recent year due to its fast growing popularity it has been extensively reverse-engineered. Protocol syntax and description are available in natural language, meanwhile a large number of compatible chatting applications have been developed, which are able to interoperate with the official MSN servers. Because of this unique situation it is worth investigating the reliability and security properties of these unofficial client implementations. In our experiment, we choose two most popular open source client applications: aMSN [1] and Gaim (now under the name pidgin [22]). Since both applications support both Linux (Ubuntu) and Windows (Windows XP) operating system, we have 4 different implementations. In the experiments we treat them as pure black-box without using any knowledge from the source code. As a matter of fact, although these applications are built on portable low level libraries, we show that they exhibit differently on Linux and Windows in terms of reliability.

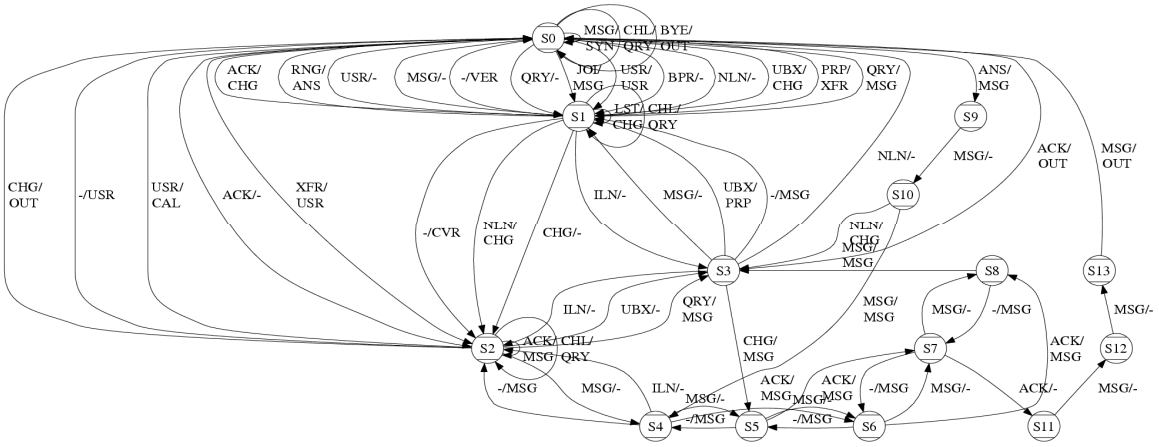

Fig. 3. Synthesize FSM Model of MSNIM Protocol Client

The syntax of the protocol is quite straightforward. A client communicates with the server sending text message over TCP connections. Each message contains a prefix specifying the type and a list of parameters. Certain types of messages can also carry a transaction ID and/or an encoded payload section. MSN protocol supports a diverse set of functionalities from simple presence information and text messaging to complicated features such as setting up video/audio conference. In our fuzz testing we focus on the login/logout procedure. When a user logs in from a client site, a sequence of messages is exchanged with the server to negotiate the protocol specification as well as to retrieve the user profile information such as list of contacts and their status.

B. Fuzz Testing Setup

We have implemented the passive synthesis algorithm described in this paper. In order to take over the input and output of the MSN client we developed a special SOCKS v5 proxy and forced the client to use it. The TCP connections are setup in the same way as a regular SOCKS v5 proxy, however both incoming and outgoing traffic are intercepted. Fig. 4 shows the software architecture of our modified proxy and test controller. First we develop a simple message encoder (decoder) that translates the incoming (outgoing) traffic from (to) MSN Protocol Data Unit (PDU). All incoming and outgoing messages (between client and server) are directed to the protocol specification synthesizer implementing Algorithm 1 and Algorithm 2. Incoming messages are mutated selectively according to certain test sequence selection strategy and the current fuzz operator. Finally, the goal of fuzz testing is determined automatically after mutated input messages are sent to the client.

C. Synthesized FSM Model

By monitoring the normal sessions of the MSN protocol we are able to synthesize an FSM model as illustrated in Fig. 3. The activities in these monitored sessions include login/logout, text messaging, changing of status and file transfer. We define both MSG_I and MSG_O as all valid MSN protocol messages, and the initial abstraction function α and β as the mapping of a message to its type (the three letter prefix of a message). During the construction of tree FSM it turns out that the two initial abstraction functions are deterministic, therefore A_I and A_O are the set of message types. The tree FSM contains 28 input symbols, 14 output symbols and 94 states. Minimizing it only takes 93 milliseconds with a Java implementation of Algorithm 2 running on a commodity PC, generating an FSM with 14 states and 48 transitions. The transition coverage criterion on this model is used to determine which input message to fuzz. One natural question is that whether this synthesized model could be correlated with the design of MSN protocol. Although it is difficult to obtain authorized formal documentation of the protocol, we are making an attempt to develop one independently from natural language description for an evaluation of the synthesized model.

D. Fuzzing Goal and Operators

The goal of testing MSN client implementations is to find an observable failure state, i.e. a crash. In terms of our formal model in Section II.B, the *GOAL* predicate checks whether the output includes a special "crash" symbol. A crash is manifested in various ways on different platforms, and, consequently, the implication for security also varies. For instance on Windows such a failure will bring up an error message box that indicates a crashed process while on Linux the crashing of process is directly exploitable by attackers. We define a crash instance as an input sequence that produces an output sequence satisfying GOAL. As far as black-box testing is concerned, a crash instance represents an error.

We developed 26 single message fuzzing functions, which fall into four categories: (1) Fuzzing data fields; (2) Fuzzing message type; (3) Intra-session message reordering; and (4) Transition substitution. The constructed FSM formal model guarantees that the fuzz functions achieve a high coverage. We further elaborate these four categories below.

- **Data Field Fuzzing (16 functions):** Functions belonging to this category modify one or more data fields of an input message and form a new one (in MSG_I) that is not acceptable

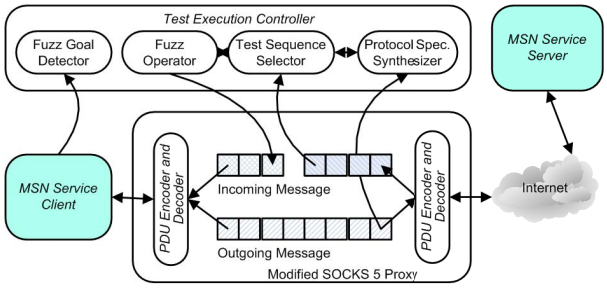

Fig. 4. Software Architecture of a MSN Client Fuzz Testing Tool

to a normal client The modifications include deleting some or all fields, inserting and repeating fields, and altering the value of a field. A single field is altered depending on its type: for a string literal we expand it to an extremely long string; for a numeric field we change its sign or decimal point; and, similarly, for E-mail address, URL or IP address types we choose suitable functions.

- **Message Type Fuzzing (4 functions):** Two fuzzing functions in this category simply change the message type to a different (defined or undefined) one. Other functions repeat or remove the message type field in a message. Since the new message is not likely to be a legitimate one of the modified type, the implementation might fail when the transition corresponding to the new type is triggered.
- **Intra-Session Message Reordering (3 functions):** This type of fuzzing functions twists the order of normal message exchange with inserting, repeating, or dropping an intercepted message within the same session. Unlike the previous two categories these function do not generate illegitimate messages, instead, they force an unexpected transition.
- **Transition Substitution (3 functions):** We also substitute an input message with another one that is likely to trigger a different transition that goes from the current state (in M_X) to another state. It seems that these functions should not cause a failure because the client is expecting the new transition. However there are two issues. First, M_X is not accurate and, as discussed, it may contain artificial paths. In fact, failures caused by transition substitution also provides an opportunities to refine M_X. Second, due to the assumption of fuzz testing we may not be able to construct an arbitrary input message. In our implementation we take advantage of the messages monitored during synthesis phase by storing and reusing the similar ones.

E. Result and Analysis

After the model is synthesized and the fuzzing functions are developed, we run our experiments on all four implementations guided by the transition coverage criteria. The test execution procedure (login/logout phase of the protocol) is fully automated and each pass attempts to cover a new transition in the model. We analyze the result in this section.

The first encouraging finding is that model based fuzz testing is very effective; almost all fuzzing functions find crash

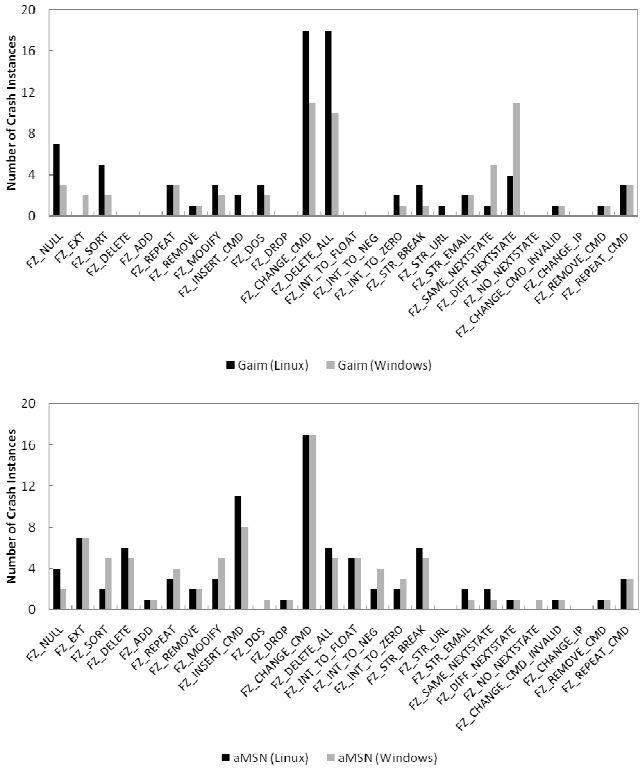

Fig. 5. Number of Crash Instances from Fuzzing

instances. Specifically, we find a total of 89 instances from aMSN and 61 instances from Gaim. We emphasize again that the instances do not necessarily correspond to individual bugs in the client application; instead, each of them represents a unique way of crashing the protocol implementation. On the other hand, we believe such result is extremely helpful for protocol testing process, especially given its high level of automation. We also show the number of instances found by each function in Fig. 5. Overall aMSN exhibits consistent behavior on Linux and Windows, while the two versions for Gaim are quite diverse. Further analysis on this issue will require software engineering efforts and is out of the scope of this paper.

We also measure the progress of finding new crash instance with the increase of transition coverage metric. The relationship for two of the functions *FZ_CHANGE_CMD* and *FZ_INSERT_CMD* is shown in Fig. 6. Since each pass of execution guarantees to test a new transition, it is not surprising to see such a ladder-shaped progress. Note that only about 35% of the transitions have been covered since the synthesized model is not restricted to login/logout phase of the protocol. Even though we have the model and hence know exactly which abstract input symbol (in A_I) can trigger a new transition, we cannot construct the concrete message (in MSG_I) due to our assumptions.

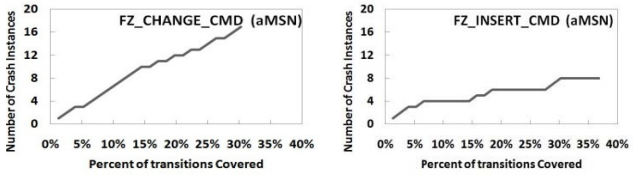

Fig. 6. Number of Crash Instances and Transition Coverage

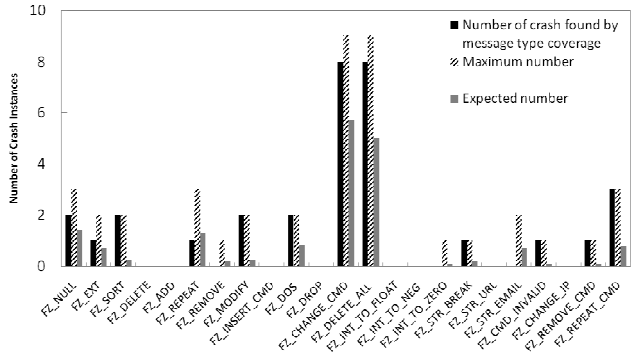

Fig. 7. Maximum, Expected and Actual Number of Crash Instances in Gaim (Windows) under Complete Message Type Coverage

Another major finding from our experiments is that different transitions in M_T with the same input symbol might behave differently in terms of finding crash instances under the same fuzzing function. In MSN protocol this phenomenon is quite common since certain types of messages can serve for multiple purposes and are, therefore, processed by different logics (source code): some are immune to a fuzzing function while others are not. The messages with different roles are exactly reflected in M_X by multiple transitions with a same input symbol. This finding shows the synthesized behavioral model is truly useful and also justifies our transition coverage criteria.

The above observation has motivated our further comparison between the flaw detection capability of our model based approach and syntax based methods such as message type coverage that is frequently used in practice. Message type coverage measures the percentage of different protocol messages that is fuzzed by a function during an experiment. Since it does not need protocol specification (only the grammar) it can be easily implemented. In our study we also used message type coverage as a guide on all four client implementations. That is, for each pass of execution, the first occurrence of a message with uncovered type is fuzzed, and then the type is marked as covered.

As expected, the experiment guided by message type coverage detects less crash instances than our experiment using transition coverage of FSM model. For a message type that corresponds to multiple transitions in the model, any one transition can be selected causing that type to be marked. Consequently, the actual vulnerable transition might be missed. Given the synthesized FSM model M_X we can calculate approximately the maximum and expected number of crash

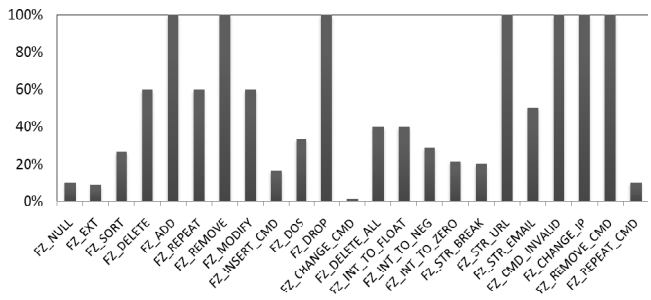

Fig. 8. Probability of Detecting Maximal Number of Crash Instance from Message Coverage.

instances detected by achieving complete message type coverage. Let t=<s_1, s_2, i, o> be a transition in M_X, where s_1, s_2, i and o are the starting state, ending state, input and output of the transition, respectively. Let T be the set of all transitions and T^* be the subset that finds crash instance for a certain fuzzing function f. When message type coverage is completed, exactly one transition of each input message type is fuzzed. The maximal number of crash instance is

$$Max_{MT} = |\{a \in A_I | \exists t < s_1, s_2, i, o > \in T^* \wedge i = a\}| \leq |T^*| = Max_{TR}$$

If we further assume that all transitions with same input type have the same probability to be covered, we can also derive the expected number of crash instances and the probability that one experiment detects a maximal number of instances as follows:

$$Exp_{MT} = \sum_{a \in A_I} \frac{|\{t < s_1, s_2, i, o > \in T^* | i = a\}|}{|\{t < s_1, s_2, i, o > \in T | i = a\}|}$$

$$Prob_{MT} = \prod_{a \in A_I} \frac{|\{t < s_1, s_2, i, o > \in T^* | i = a\}|}{|\{t < s_1, s_2, i, o > \in T | i = a\}|}$$

We note that the formula above is not precise since it does not consider the topology of M_X, since atransition in M_X may never be reached without reaching another transition. If an implementation is always executed from the initial state, the actual number of crash instances can be different than the theoretical result. As a matter of fact, Both Exp_{MT} and $Prob_{MT}$, which we calculate, are lower bounds. We measure the actual numbers by complete message type coverage for one implementation (Gaim on Windows), as shown in Fig. 7. For about half of the functions some crash instances are missed, and it clearly shows the inefficiency of such syntax-based coverage criteria. Based on the synthesized model of Fig. 3, we also calculate $Prob_{MT}$ for each function as shown in Fig. 8. For the majority of functions, the probability of detecting all crash instances is below 60%. Our experiments prove the advantage of using a behavioral model over simple syntax based test selection schemes. On the other hand, transition coverage is not the only (nor the most exhaustive) criteria one can use on an FSM model. Alternatives such as path coverage can be more effective under certain circumstances.

VI. RELATED WORKS

We briefly discuss existing works, which are closely related to ours. The idea of using input fuzzing to detect flaws of communicating systems has been investigated for more than 20 years [3] and white-box approaches have been predominant. Many advanced theory and techniques such as symbolic execution [9][10] and binary interception [14] have been applied to find input data causing failures. Existing works on black-box fuzzing are mostly ad-hoc and each is coupled with a specific protocol. The PROTO project [15] manually developed fuzz test suites for many popular Internet protocols for an evaluation. In recent years there is a number of open source network fuzzing tools, covering a wide spectrum of protocols, including TCP/IP, HTTP, P2P and many others. Most of these tools hardcoded knowledge of target protocols and provided limited supports for automatic input selection.

Synthesized mathematical models of protocol implementation have been used by many researchers. In [4] the authors proposed to use predicate formula to characterize implementations and to further use it for error detection and protocol fingerprinting. Our application of the model is also for the purpose of deriving input sequences with comprehensive coverage. Other works such as [7] and [18] aimed at reverse engineering of protocol message format using a variety of learning mechanisms. These results to a large extent enable us to develop protocol message encoder and decoders. On the other hand, the protocol synthesis technique studied in this paper is quite different than the works of of synthesizing high quality protocol implementation from design models [26]. Finally, our algorithm for tree FSM reduction is based on the classical results described in [5] [12] [20], and the active synthesis approach uses the theory of machine learning [2] and black-box checking [21].

VII. CONCLUSION AND FUTURE WORK

This paper proposes a model-based methodology for detecting security and reliability flaws of network protocol implementations. By using an automatically synthesized formal protocol behavioral model to guide input selection, our method changes the random and manual nature of black-box protocol security testing that is considered a significant drawback. Using protocol fuzzing as a case study, we conduct extensive experiments with different implementations of MSN IM client. The result proves our approach is more effective than the existing syntax based security testing methods.

It is a challenge to have a more generic and powerful flaw detection framework. A key problem is how to improve the quality of the synthesized specification. As already discussed in the paper new traces are monitored during testing and these traces can be utilized to refine the model M_X. Particularly, since these input sequences are different than the ones from a regular protocol component, they are very useful for eliminating the invalid paths from M_X. We plan to integrate the testing feedback into model synthesis, forming essentially a close loop process.

As far as practical aspects of our fuzzing tool are concerned, we plan to gain more insights of the crash instances and investigate more coverage metrics and fuzzing schemes. Our current tool uses a family of built-in fuzzing functions. It is beneficial to decouple and reuse the design of those functions for different protocol and even different formal models. Finally, the passive protocol synthesis approach proposed in this paper is generic and can be adopted in other different applications than fault discovery. For instance, reverse engineering of legacy protocol system is of great practical interest. In our future work we plan to investigate the synthesis techniques for models with richer semantics, such as extended FSM.

REFERENCES

[1] The Amsn Project. http://www.amsn-project.net/

[2] D. Angulin. Learning regular sets from queries and counterexamples. Information and Computation, 75, pages 87-106, 1987.

[3] J. Barton, E. Czeck, Z. Segall and D. Siewiorek. Fault Injection Experiments Using FIAT. IEEE Trans. on Computers, 39(4), pages 575-582, 1990.

[4] D. Brumley, J. Caballero, Z. Liang, J. Newsome and Dawn Song. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. in Proceedings of USENIX Security Symposium, 2007.

[5] A.W. Biermann and J.A. Feldman. On the Synthesis of Finite State Machines from Samples of Their Behavior. IEEE Trans. on Computers, vol. 21, pages 592-597, 1972.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. MIT Press, 1990

[7] W. Cui, J. Kannan and H. Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. The 16th USENIX Security Symposium, 2007.

[8] D. Dolev and A. Yao. On the security of public-key protocols. IEEE Transaction on Information Theory 29, pages 198-208, 1983.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 213--223, 2005.

[10] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. Technical Report MS-TR-2007-58, Microsoft, 2007.

[11] L. Gordon, P. Loeb, W. Lucyshyn and R. Richardson. 2005 CSI/FBI Computer Crime and Security Survey. Computer Security Institute, 2005.

[12] S. Gören and F. J. Ferguson. On state reduction of incompletely specified finite state machines. Computers and Electrical Engineering, Vol. 33(1), pages 58-69, 2007.

[13] M. Howard. Inside the Windows Security Push. IEEE Security & Privacy, pages 57-61, 2003.

[14] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. USENIX Windows NT Symposium, pages 135-143, 1999.

[15] R. Kaksonen, M. Laakso and A. Takanen. System Security Assessment through Specification Mutations and Fault Injection, the IFIP International Conference on Communications and Multimedia Security Issues of the New Century, 2001.

[16] D. Lee and K. Sabnani. Reverse Engineering of Communication Protocols. In Proceedings of IEEE ICNP, 1993.

[17] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In Proceedings of the IEEE, pages 1090–1123, 1996.

[18] Z. Lin, X. Jiang, D. Xu and X. Zhang. Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution. In Proceedings of 15th Network and Distributed System Security Symposium (NDSS),2008.

[19] P. Oehlert. Violating Assumptions with Fuzzing. IEEE Security & Privacy, pages 58-62, 2005.

[20] A. Oliveira and S. Edwards. Limits of Exact Algorithms for Inference of Minimum Size Finite State Machines. The 7th International Workshop on Algorithmic Learning theory pages 59-66, 1996.

[21] D. Peled, M. Y. Vardi, M. Yannakakis. Black-box checking, In Proceedings of IFIP FORTE/PSTV, 1999.

[22] The Pidgin Project. http://www.pidgin.im/
[23] G. Shu, Y. Hsu and D. Lee. Detecting Communication Protocol Security Flaws by Formal Fuzz Testing and Machine Learning. FORTE 2008. LNCS, vol. 5048, pp. 299-304, 2008.
[24] G. Shu and D. Lee. Testing Security Properties of protocol implementations – a machine learning based approach. In Proceedings of IEEE ICDCS, 2007.
[25] US-CERT Vulnerability Notes: http://www.cert.org/
[26] K. Zeroual, M. Yassini. A protocol synthesis algorithm: a relational approach. In Proceedings of IEEE ICNP,1995.