Detecting Communication Protocol Security Flaws by Formal Fuzz Testing and Machine Learning

Guoqiang Shu, Yating Hsu and David Lee

Department of Computer Science and Engineering, the Ohio State University
Columbus, OH 43210, USA
{shug,hsuya,lee}@cse.ohio-state.edu

Abstract: Network-based fuzz testing has become an effective mechanism to ensure the security and reliability of communication protocol systems. However, fuzz testing is still conducted in an ad-hoc manner with considerable manual effort, which is mainly due to the unavailability of protocol model. In this paper we present our ongoing work of developing an automated and measurable protocol fuzz testing approach that uses a formally synthesized approximate formal protocol specification to guide the testing process. We adopt the Finite State Machine protocol model and study two formal methods for protocol synthesis: an active black-box checking algorithm that has provable optimality and a passive trace minimization algorithm that is less accurate but much more efficient. We also present our preliminary results of using this method to implementations of the MSN instant messaging protocol: MSN clients Gaim (pidgin) and aMSN. Our testing reveals some serious reliability and security flaws by automatically crashing both of them.

Keywords: Fuzz testing, Security Testing, Protocol Synthesis.

1 Motivation

Network-based fuzz testing is a very effective approach to improve the security and reliability of protocol system implementations [7, 9]. It works by mutating the normal traffic at the ingress interface of a component in order to reveal unwanted behavior such as crashing or confidentiality violation [3]. Identifying such flaws is extremely important since they might be exploited by malicious parties to launch attacks. On the other hand, it has been reported that for today's complicated system these flaws are ubiquitous due to incorrect assumptions on the input data. However, unlike software fuzz testing where white-box approach [4, 5] is widely used, protocol fuzz testing is usually conducted in an ad-hoc manner with input selected either randomly or manually [5]. With such restrictions it is very difficult to measure the comprehensiveness of testing and the level of test automation is low. With knowledge of the protocol message format, some preliminary systematic approaches such as message type covering become feasible. However they are in general inaccurate for various reasons. For example, messages of same type could serve very different roles in a protocol session and therefore should be distinguished in testing.

In this work we propose an automated solution to improve the quality and measurability of black-box fuzz testing using a formal protocol specification synthesis approach. The key idea is to obtain an approximate formal model of the component under test and use it to automatically guide test selection for better fault coverage. Such model is based on presumed knowledge of protocol messages [2] while its primary function is to describe the states and transitions in a session. Note that formal

specifications are usually not available in practice for real protocol systems. Construction of such a specification model endows significant guidance to systematic black-box testing which is otherwise impossible. Specifically, we can design test sequences to achieve formally defined fault coverage criteria with regard to the specification, and meanwhile to intelligently choose mutated inputs based on special context in the model. Note that the specification synthesis problem we study has fundamental difference with the protocol design and implementation synthesis problems extensively studied in the literature, which aim at using formal models to facilitate developing error-free protocol instead of recovering the specification for given implementation.

In this paper we discuss two alternative formal methods for Finite State Machine (FSM) based specification synthesis which we applied in fuzz testing of real world network applications. Our experiments show that this approach is promising in automatic discovery of new bugs in protocol implementations of real internet applications.

2 Formal Protocol Synthesis

We adopt a variant of classic Communicating Extended Finite State Machine (CEFSM) to model a communication protocol. The behavior of each protocol principal is described by a deterministic EFSM that has state variables and input/output message parameters with symbolic value domain. The detailed modeling is in [8, 11]. In this work we focus on one principal and use its reachability graph (an FSM) representation. An *FSM* is a 5-tuple $<S,s_0, I, O, f_{next}, f_{output}>$, where S and s_0 are state (configuration in EFSM) set and initial state, I and O are input and output alphabet, $f_{next} : S \times I \rightarrow S$ is the transition function and $f_{output} : S \times I \rightarrow O$ is the output function. Both f_{next} and f_{output} might be partial function. We call an FSM a tree FSM if its state transition graph is a tree. A trace of FSM is a sequence of input/output pairs, $tr = \{<I_1,O_1>,<I_2,O_2>,\ldots,<I_k,O_k>\}$, and a test case is simply a sequence of inputs.

Given a black box protocol component implementation B, our objective is to synthesize a deterministic FSM model M_x that later guides our fuzz testing algorithm. M_x should ideally be an abstraction of all observed behavior of B, and its input (output) alphabet I_x is a subset of B's input alphabet I_B. The ultimate goal of testing then is to find a sequence of any length L: $\{<I_k,O_k>, 0 \le k \le L, I_k \subseteq I_B, O_k \subseteq O_B\}$ that will lead B to an observable failure state. Below we discuss two approaches to construct M_x – an active learning algorithm and a passive machine minimization algorithm.

2.1 FSM Learning Algorithm

Since the tester has full control of the input and output of B, an obvious way to get its model is through active learning. Following the theoretical insights of [1,10] on automata learning, we design the following procedure. An estimation model B^* of the implementation B is maintained and initialized as an FSM with an initial state only. B^* is updated as more traces are discovered according to the supervised FSM learning algorithm L^*_{fsm} (based on Angluin's L^* algorithm with details omitted due to space limit; see [1]). A conformance test generator serves the role of "teacher" in learning process that provides traces as counter-example – showing the difference between B^* and B. The counter-example is used to prepare for the next estimation that becomes supposedly more accurate: containing more input types or more states.

This iterative process starts with a small subset of input alphabet and terminates when the teacher is not able to find any counter-examples to help learning. We can prove this strategy is always "promising" in the following sense: if B contains N states and P inputs, at most $(N+P)$ guesses will be made before we get $M_x=B^*=B$. The cost of this process is determined by both the strategy used by the teacher and the L^*_{fsm} learning algorithm itself. We could prove that it takes $O(P^*\cdot N^{*2})$ to update B^* with P^* inputs and N^* states, and the total cost of learning B in worst case is $O(T\cdot P^2\cdot N^2+T\cdot P\cdot N^3)$ where T denotes the cost of calculating the counter-example at each round. In practice due to this high cost we usually stop after several iterations with an approximate model.

2.2 Partial FSM Minimization Algorithm

We also study an alternative that requires more observation but potentially less computation. The idea is to first gather a large number of traces from B by *passive* monitoring, compute a tree FSM, and minimize it. Given a set of traces, the synthesis of tree FSM is quite straightforward. Starting with empty FSM we add one trace at a time. We find the longest prefix of a trace that is already in the current FSM, presumably ending at state s, then create a new branch from s with the rest of the trace. One practical issue in this step is handling session related fields. We want to identify data fields in an input/output message whose value does not affect the state transition of this session and therefore could be symbolized. Typical examples of such fields include username, nonce and session ID. Identification of these fields reduces the redundancy of the tree FSM; however it is nontrivial and sometimes requires manual effort. In our on-going work we are investigating efficient and automated solutions.

After the tree FSM is constructed, we want to minimize the number of states by merging compatible sets. Minimization problem for partial FSM is a well studied NP-hard problem and many heuristic solutions have been proposed [6]. A simple optimistic algorithm is Bierman's algorithm also described in [6]: first a set of constraints of merging is calculated dictating which pair of states can be merged and which two pairs must be both merged or both unmerged; after that new state IDs are given to the states and whenever a constraint is violated the assignment is modified. The complexity of this algorithm in worst case is obviously exponential but we can modify it by limiting backtracking to get a polynomial suboptimal algorithm. As an extreme case, we might choose not to backtrack at all (i.e. always assign new state ID) to achieve linear time.

3 Fuzz Testing Strategy

Once we have synthesized the approximate protocol specification M_x, it is used to guide fuzz testing experiments. Coverage metrics can be formally defined to measure the comprehensiveness of a set of tests. Let $I_0I_1\ldots I_L$ be a sequence in M_x, and a fuzz testing sequence has the general form of $I_0I_1\ldots I_k\, f_{fuzz}(I_{k+1}\ldots I_L)$, where the prefix of length k is a leading sequence that takes B to a certain state and the rest is the result of applying a fuzz function $f_{fuzz}:I^*\rightarrow I_B{}^*$ to the original postfix. Let us consider a special function that modifies the format of last input message only, and we want to test this function for all transitions in M_x. Given a set of K test sequences $\{SEQ_i = PREFIX_i\, f(LAST_i) \mid 0\le i\le K, PREFIX_i\in I^+,\ LAST_i\in I\}$, the formula below computes its transition

coverage as the number of transitions covered by the last input of a message divided by the total transitions in M_x.

$$TR\_Coverage = \frac{\left|\{< s', LAST_i > | s' = f_{next}(s_0, PREFIX_i) \wedge f_{next}(s', LAST_i)\downarrow, 0 \le i \le K\}\right|}{\left|\{< s, i > | f_{next}(s, i)\downarrow, s \in S, i \in I\}\right|}$$

Other metrics could be similarly developed corresponding to popular fuzz functions on input sequence such as repetition, stealing, replay and pre-play. The actual test generator should be designed to give preference to sequences that increase the metric.

4 Experiments and Evaluation

We have implemented both of the synthesis strategies as well as several typical fuzz functions and applied them to evaluate two popular alternatives of MSN instant messaging clients Gaim (pidgin) and aMSN. In order to take over the I/O of the client we developed a proxy through which the client is connected to the server (shown in Figure 1). We also implement a simple encoder and decoder for MSN protocol messages. The goal of fuzz testing is to find input sequences that will crash the client process (a behavior that is definitely unwanted). We synthesize a model for the login phase of MSN protocol containing around 50 states and 70 transitions. Several typical fuzz functions on single transition are manually developed, after which testing is done automatically toward 100% transition coverage for each function.

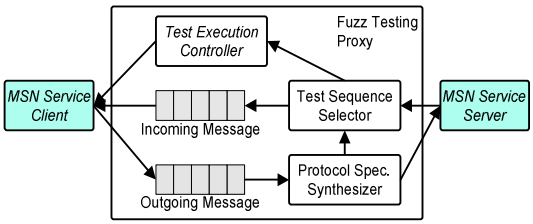

Fig.1. An MSN Client Fuzz Testing Tool

Table 1. Testing two MSN Clients with synthesized model

Size of tree FSM	450
#states/#transitions of synthesized FSM	50/ 70
Fuzz functions used	5
Bugs found in Gaim	3

As summarized by Table 1, we found many previously unknown bugs of both clients and we are continuously uncovering more. Our fuzz functions fall into two categories: (1) changing the data field of a message to form an invalid input from I_B-I_{Mx}; and (2) changing the message type to form an undefined transition with respect to the current state. Below we report instances of bugs from each category.

- **Invalid Status Code:** ILN message type is used for buddy presence notification; the syntax of the command is "*ILN TrID status_code Account Display ClientID*" where the *status_code* field is used to indicate the presence of a contact such as available, busy, or away. MSN protocol gives a list of legitimate status codes but if we change it to an invalid value, aMSN crashes immediately after receiving the message.
- **Elimination of E-Mail Address Field:** a simple fuzz function that eliminates every field from an input message that is in the form of an email address (used as account name) will cause both Gaim and aMSN to crash.
- **Skipping Contact List Message:** in order to obtain the buddy list of a user, a sequence of messages is exchanged between the server and client including an *LST*

message to download the contact list followed by a sequence of *ILN* messages to obtain presence information. We found that if the LST message is skipped (i.e. dropped); aMSN will crash when receiving the *ILN* message.

- **Random Message Type Mutation:** we could simply modify the input message type to a random message type that is undefined in the current state. For instance, when we change *CVR* or *VER* message (both used to negotiate protocol version) to *LST* type, both clients will crash. A variant of this operation is random message type swapping of two adjacent transitions, and aMSN crashes when this is applied to *LST* and *UBX* messages.

5 Conclusion

We investigate the proposed fuzz testing approach that has shown great potential and practicality. A key problem to tackle is how to improve the quality of the synthesized specification. For instance, our current tool does not recover the dependency relationship among message fields, which gives valuable insights regarding what input might be destructive. Toward this goal analysis for both control plane and data plane of the protocol traces are to be integrated. On the other hand, we envision that formal protocol synthesis techniques could be useful in other domains, such as protocol reverse engineering [2] and network testbed development [12].

References

1. D. Angulin. Learning regular sets from queries and counterexamples. Information and Computation, 75, pages 87-106, 1987.
2. W. Cui, J. Kannan and H. Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. The 16th USENIX Security Symposium, 2007.
3. D. Dolev and A. Yao. On the security of public-key protocols. IEEE Transaction on Information Theory 29, pages 198-208, 1983.
4. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation), pages 213--223, 2005.
5. P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. Technical Report MS-TR-2007-58, Microsoft, May 2007.
6. 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
7. M. Howard. Inside the Windows Security Push. IEEE Security & Privacy, pages 57-61, 2003.
8. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In Proceedings of the IEEE, pages 1090–1123, 1996.
9. P. Oehlert. Violating Assumptions with Fuzzing. IEEE Security & Privacy, pages 58-62, 2005.
10. D. Peled, M. Y. Vardi, M. Yannakakis. Black-box checking, In Proceedings of IFIP FORTE/PSTV, 1999.
11. G. Shu and D. Lee. Testing Security Properties of protocol implementations – a machine learning based approach. In Proceedings of IEEE ICDCS 2007
12. L. Wang, C. Ellis, W. Yin and D. D. Luong. Hercules: An Environment for Large-Scale Enterprise Infrastructure Testing. in Proceedings of the Workshop on Advances and Innovations in Systems Testing, 2007