Can You Fool Me?
Towards Automatically Checking Protocol Gullibility

Milan Stanojevic *UCLA*
Ratul Mahajan *Microsoft Research*
Todd Millstein *UCLA*
Madanlal Musuvathi *Microsoft Research*

Abstract – We consider the task of automatically evaluating protocol gullibility, that is, the ability of some of the participants to subvert the protocol without the knowledge of the others. We explain how this problem can be formalized as a game between honest and manipulative participants. We identify the challenges underlying this problem and outline several techniques to address them. Finally, we describe the design of a preliminary prototype for checking protocol gullibility and show that it can uncover vulnerabilities in the ECN protocol.

1. INTRODUCTION

Modern communication protocols are regularly used among entities that should not trust each other to faithfully follow the protocol. Sometimes trust is unwarranted because a participant may not know the identity of the other entities. Even with known identities, there is still the possibility that others have been hijacked by malicious agents. Further, an entity can inadvertently violate the protocol due to bugs in the implementation.

Whatever the cause, protocol violations can break important protocol invariants, leading to a variety of attacks. For example, Savage et al. show that a TCP receiver can fool the sender into sending faster than the intended congestion controlled rate [19]. In fact, they point out three different ways to accomplish this task. Spring et al. show that an ECN (Explicit Congestion Notification) receiver can fool the sender into ignoring congestion by simply flipping a bit [21], undermining the central purpose of the protocol. Researchers have reported similar kinds of vulnerabilities in peer-to-peer protocols, multi-hop wireless networks, and routing protocols as well [4, 14, 1].

Thus, it is strongly desirable that communication protocols be robust against accidental or intentional manipulation by participants. We say that a protocol is *gullible* if a subset of the participants can violate desirable properties of the protocol by disobeying the protocol in some fashion. Today, protocol gullibility is determined through manual inspection; for example, all vulnerabilities above were discovered manually. While significant progress has been made recently in identifying bugs in protocol implementations [7, 17, 11], bug-free honest participants may still be fooled by manipulative participants, as the examples above illustrate.

In this paper, we pose the problem of automatically evaluating a protocol's gullibility. To our knowledge, prior work has either detected gullibility manually or has automatically checked for specific vulnerabilities in security protocol specifications [13, 15, 20, 18, 5]. We define the general problem of protocol gullibility detection, discuss the challenges involved, and propose techniques that can help to address these challenges. Finally, we report on the design of a preliminary prototype gullibility checker along with some initial results on a very simple protocol.

We formalize the problem of automatic gullibility detection in Section 3 as a game between an angelic component that consists of honest players and a demonic component that consists of manipulative players. The angelic component follows the protocol, while the demonic component can take any action, including sending or not sending any particular packet. A protocol is considered gullible if there exists a strategy for the demonic component such that a desirable property of the protocol is violated.

Automatically determining protocol gullibility is challenging. The key challenge is the enormity of demonic strategy space. At any step, any particular bit pattern can be sent, and complicated strategies may involve a long sequence of particular packets. A second difficulty is the need to search the space of network conditions, because some strategies succeed only under certain network conditions. Finally, even determining when a strategy has succeeded may be difficult, requiring comparison with a reference run where all participants are honest. Section 4 describes these challenges in detail and proposes several techniques to address them.

To begin exploring the problem and our approach, we have developed a preliminary gullibility checker that incorporates a subset of the techniques we propose. We present initial results from using our checker to analyze an implementation of the ECN protocol. The checker can successfully discover the attack that was previously manually discovered [21] as well as some variations on this attack. Our results for ECN, a very simple protocol, are encouraging and so we are excited to continue exploring our approach on more complex protocols.

2. EXAMPLE MANIPULATIONS

We describe below some example vulnerabilities found in existing protocols, to provide insight into the kinds of manipulations that we are interested in uncovering.

ACK division in TCP [19]: TCP increases its congestion window in units of entire packets whenever a packet that acknowledges previously unacknowledged bytes ar-

rives. A manipulative receiver can speed transfers by acknowledging individual bytes rather than entire packets (as an honest receiver would).

Duplicate ACKing in TCP [19]: A TCP sender sends more data in response to any acknowledgment, since the acknowledgment signals that some data left the network. A manipulative receiver can speed transfers by generating multiple, duplicate acknowledgments for the last received sequence number.

Optimistic ACKing in TCP [19]: TCP assumes that the time between a data segment being sent and an acknowledgment being received is a round trip time, and its congestion window increases as a function of that time. A manipulative receiver can speed transfers by optimistically sending acknowledgments for packet sequence numbers that have not arrived yet.

Hiding congestion in ECN [21]: In the ECN protocol, routers set a bit in the header to signal congestion. The receivers then reflect this bit when sending packets to the sender, at which point the senders reduce their transfer rate. A manipulative receiver can speed transfers by setting the congestion bit to zero instead of reflecting it.

Dropping packets in multi-hop wireless networks [14]: Multi-hop wireless networks enable connectivity by having nodes relay for one another. Manipulative nodes can simply drop all packets that they are asked to relay while reaping the benefits by having others relay for them.

Lying about connectivity in routing protocols [1]: Routing protocols such as OSPF and BGP rely on nodes accurately reporting their connectivity, i.e., who they connect to and the connection cost. Manipulative nodes can significantly distort routing paths by lying.

Attacks in DHTs [4]: Castro *et al.* present a variety of ways in which a manipulative node can hurt the overlay. The set of possible manipulations is large. It includes how the node identifiers are generated, how routing messages are propagated, and how messages are forwarded.

All the vulnerabilities above were discovered manually. We want to automate the search for these kinds of vulnerabilities. Researchers have proposed fixes to these vulnerabilities. Automated tools can also help determine if the fixes are themselves robust.

3. PROBLEM STATEMENT

Consider a communication protocol that two or more parties use in order to achieve a common goal. The honest participants execute the protocol correctly. The manipulators have complete freedom in what they choose to send (or not send) and when. Multiple manipulators may also collude. We seek to determine if the protocol is gullible, that is, the manipulators can prevent the honest participants from achieving the goals of the protocol.

We formalize this problem as a two-player game between A and D, the *angelic* and *demonic* components. A consists of honest players. These players communicate with each other and with D by exchanging messages. D consists of manipulative players, possibly colluding through out-of-band mechanisms. Finally, assume that there is a property P over the state of A and D that represents an invariant that the protocol should never violate.

The game proceeds by alternating moves of the two components. On its turn, A chooses one of the possible actions allowed by the protocol, such as a packet receive or a timer event. The choices available to A represent the nondeterminism that is outside D's control. In particular, actions of the network, e.g., packet losses, are considered moves of A. D responds by consulting its attack strategy. Its move involves either sending an arbitrary packet to A or choosing not to respond. The protocol under study is gullible if there exists a strategy for D with which D can eventually drive the system to a state that violates P.

Our problem formulation has two notable features. First, we distinguish between angelic and demonic nondeterminism, because different methods are required to systematically search over them. Given a state, the angelic component usually has a handful of moves that obey the rules of the protocol. The demonic component can, however, send an arbitrary packet, representing an astronomically huge search space at each step. Moreover, the demonic component might use a stateful strategy, whereby the choices made at one step depend on the preceding choices. In contrast to our formulation, the current work on model checking system implementations [7, 17, 11] has either no demonic component or a very restricted one.

Second, the property P that identifies when manipulation has happened depends on the protocol under test. As a simple example, P can state that the connection queue should not be able to remain full forever, which represents a denial-of-service attack. A more complicated example property would limit the throughput that can be obtained by the demonic component. Checking for violations of this kind of property is important, since many discovered manipulations concern resource allocation attacks [19, 21, 14, 4]. For such cases, we propose that P be specified in terms of a comparison to a "reference" behavior that occurs if the demonic component were to honestly follow the protocol. For example, congestion control protocols could require that no receiver gets more data than what it would get by following the protocol; DHTs and wireless relaying protocols could require that the ratio of packets relayed and generated not change; and routing protocols could require that forwarding table pointers not change. More generally, we could specify that important state variables for a protocol not change.

4. CHALLENGES AND TECHNIQUES

We now describe the challenges in building a practical tool to test protocol gullibility and propose techniques to address these challenges.

4.1 Challenge: Practical Strategy Search

The primary challenge that we face is that the space of possible demonic strategies is huge. There are 2^{12000} possibilities for a 1500-byte packet that the manipulator can send. Further, that is just the size of the search space for a single message; complicated attacks may depend on sending a particular sequence of packets.

We propose a variety of techniques to reduce the search space size. These techniques leverage the structure inherent in network protocols as well as some properties that are common across large classes of protocols. While these search-space reduction techniques can potentially cause our tool to miss some attacks, we believe that many vulnerabilities of interest, including most of the ones described in Section 2, can still be found. Systematically discovering such vulnerabilities would be a significant improvement in the current state of art and is a first step towards automatically discovering other kinds of attacks.

• *Consider only the header part of the packet* The control flow of most protocols is based on the contents of the header and not on the payload. Hence, instead of considering the entire packet, we focus on headers. Within a packet, what is header versus payload depends on the protocol being tested. For instance, for IP everything other than the IP header constitutes the payload even though that payload may contain TCP headers. This technique enormously reduces the search space. If the header size is 40 bytes, for 1500-bytes packets, the single-step search space goes from 2^{12000} to 2^{320}.

• *Consider only syntactically correct packets* Protocol packets are not random bit strings but have specific formats. For instance, the checksum field occurs in a certain location and is computed over specific bits. Honest participants typically discard incoming packets that are not in the requisite format. Accordingly, rather than searching the space of all bit patterns, we focus on packets that comply with that format. The packet format can be provided by the user or automatically inferred [6] from traces. While there may be some attacks that involve non-compliant packets, these are very likely to be due to implementation bugs, such as buffer overflows. We are not interested in finding such bugs; other methods, such as fuzzing [16, 8], are more apt for finding them.

• *Exploit header-field independence* To further reduce the search space, we assume that most header fields are independent of one another with respect to possible vulnerabilities. That is, most vulnerabilities can be discovered through a systematic search within the possible values for groups of fields, keeping the values of other fields unchanged. For instance, for TCP one might assume that the sequence number and acknowledgment number fields are independent, allowing us to independently search the fields for possible attacks. Since each field is 32 bits, searching them together would yield a space of 2^{64} possible values, which searching them independently yields a much smaller space of $2 \times 2^{32} = 2^{33}$ values. We rely on the user to provide information about which fields should be considered independent of one another.

• *Consider only limited-history strategies* We expect that many interesting attacks require the manipulator to take only a small number of basic steps. The attack is then carried out by repeated application of this small sequence of steps. For instance, the ECN vulnerability mentioned earlier is a single-step strategy: the manipulator needs to send a specific bit pattern in order to get more bandwidth. Therefore, our tool can bound the length of strategies that it will consider to a small constant.

• *Leverage program analysis techniques* As mentioned above, we consider only syntactically correct packets, since other packets are typically ignored. However, many syntactically correct packets may also be ignored by an honest participant. For instance, if IPv4 is the protocol being checked, packets with anything but the value of 4 in the version field are ignored. As another example, TCP senders ignore acknowledgments for sequence numbers below the last acknowledged sequence number. We propose to identify the legal values of header fields using program analysis of protocol source code; our search can then ignore other values for these fields.

Program analysis can also help direct the strategy search itself by identifying conditions under which an honest participant's state can change in ways that are beneficial to a manipulator. For instance, program analysis of a TCP implementation can determine values of header fields in a received packet that cause the honest participant to send more bytes. We intend to allow the tool user to provide a set of variables in the honest participants' state, and program analysis will direct the search to strategies that cause these variables' values to change.

For both types of analyses above, we hope to leverage recent work on *directed random testing* [9], which combines *symbolic execution* [12] with program testing.

4.2 Challenge: Variable network conditions

Some protocol vulnerabilities are exploitable only under certain network conditions. For instance, the ECN bit-flipping vulnerability comes to light only if the network is congested and thus marks congestion bits in some packets. If we simulate only uncongested paths, we will not be able to uncover this ECN vulnerability.

Therefore, for each strategy considered during strategy search, we must search the space of possible network path behaviors. We make the simplifying assumption that the paths between each pair of participants are independent of one another. We can then separately characterize each such path, for instance, by its loss rate and latency. The behavior of each path defines a space of network conditions, which our tool searches for each demonic strategy.

Type	Description	Default strategies
Fixed	Fields that should not be modified	None
Checksum	Fields that represent a checksum over certain bits in the header	None
Enum	Fields that take on specific bit patterns. E.g., Protocol field in IP header	Pick a value deterministically; pick one at random
SeqNum	Fields that represent sequence numbers	Subtract or add a constant value; multiply or divide by a constant value
Range	Fields that take on a range of value. E.g., Addresses and port numbers	Pick a value deterministically; pick one at random
Id	Fields that contain identifiers. E.g., IP identifier in IP header, or node identifier in peer-to-peer protocols	Pick one at random
Other	All other fields	User-specified

Table 1: Fields types and default strategies in our system.

4.3 Challenge: Determining when a strategy has been successful

As mentioned earlier, for complex properties, particularly those related to resource allocation, a strategy's success cannot be determined simply by running the strategy; instead we must compare against reference behavior when all participants are honest. However, two individual runs of the protocol cannot be meaningfully compared directly, due to the nondeterminism in the angelic component (e.g., network conditions). For example, under a given network condition of 20% loss rate, the behavior of TCP is not unique but depends on which specific packets are lost.

Therefore, to compare a manipulated protocol against reference behavior, we run each version of the protocol multiple times under a given network condition, obtaining a distribution for each over some set of metrics (e.g., number of messages sent in a given amount of time). We can then use statistical tests (e.g., the Kolmogorov-Smirnov test or Student's *t*-Tests) to determine if the two distributions are statistically different.

5. PROTOTYPE CHECKER

To study the feasibility of automatically checking protocols for gullibility, we are implementing a prototype checker. Our current implementation uses all of the techniques described above except for program analysis. Further, rather than asking the user for information about field independence, we simply assume that all fields are independent of one another. Finally, we use network simulation to explore angelic nondeterminism.

Our system works directly with protocol implementations written in the Mace language for distributed systems [10]. We create the manipulators as modifications of an "honest" implementation of the protocol. At a high level, the manipulator has three types of actions that cover the space of possible strategies: i) drop packets that honest players would have sent; ii) modify the contents of packets that honest players would have sent; and iii) send packets when honest players would not have sent a packet.

We have created a simple API for dropping and modifying packets, along with a default implementation of the API (a C++ class). Wherever the protocol source code calls the function to send a packet, we insert a call into our API's `modify_or_drop` function, passing the packet to be sent. Our function returns the modified packet and indicates whether it should be dropped.

We have not yet implemented packet insertions at arbitrary times, the third action above. Our plan is to add a timer to the protocol implementation. An arbitrary packet can be sent when this timer fires.

The protocol tester provides three required inputs and one optional input to our system:

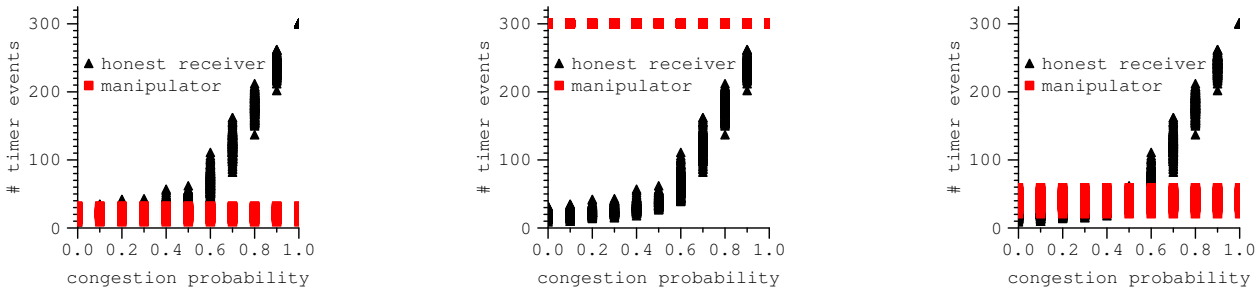
1. Network configuration This includes the number of participants, and the fraction of participants that are honest. If the protocol is asymmetric, such as TCP sender vs. receiver, the tester also specifies which aspect to test. This setup information is distinct from network path conditions (e.g., loss rate) for which we automatically explore the various possibilities. In the future we plan to automatically generate and test a range of possible setups as well.

2. Variables of interest The tester specifies the property to be tested indirectly by providing a set of variables of interest within the protocol implementation. The intent is that the tester is interested in ways in which a manipulator can cause deviations in the values of these variables. If the implementation does not already have the necessary variables, we expect the testers to implement them. For instance, while the current congestion window is an existing state variable in a TCP implementation, total traffic sent may not be, but it can be easily implemented.

3. Protocol header format The format is specified in terms of the header fields and their types. The types that we currently use are listed in Table 1. This list is based on an informal survey of several common protocols and it may grow as we experiment with more protocols. For some of the types, e.g., *Enum* and *Range*, the format must specify the possible values of their fields.

4. A packet modifier class The packet modification operation produces a modified version of the input packet. Each header field is modified independently, depending on its type. The default modification strategies are shown in Table 1. All default modification strategies are memoryless: they do not depend on what was sent before. Users can optionally provide their own packet modifier class implementing our API, in order to specify their own strategies for these and other field types.

Given these inputs, we use the simulation engine provided by the MACE framework [11] to evaluate protocols.



(a) Set bit to 0 (b) Set bit to 1 (c) Set bit randomly

Figure 1: Results of using our prototype to check the gullibility of the ECN protocol. Each graph corresponds to a different cheating strategy by the receiver. The x-axis indicates the fraction of packets on which the congestion bit was set. The y-axis is the number of timer events to deliver 300 packets to the receiver.

In each run of the simulator, we fix a particular set of network path conditions as well as a particular modification strategy for each header field. The cross product of network path conditions and possible modification strategies thereby forms the state space that our engine explores.

For each pair of a set of network conditions and a modification strategy, we simulate the protocol multiple times. We also simulate multiple runs of a completely honest version of the protocol for each possible set of network conditions. If there are statistically significant differences in the values of variables of interest between the honest and manipulated runs for a given strategy and set of network conditions, we deem the protocol to be gullible.

6. CASE STUDY: ECN

We now present results from using the preliminary version of our tool to test the ECN protocol. We chose to start with ECN because of its simplicity. Nonetheless, a study of even this simple protocol reveals many relevant insights and provides initial evidence for the feasibility of automatic gullibility checking.

We implemented a version of the ECN protocol in the Mace framework. Its header has only one field, of type Enum with 0 and 1 as the possible values. The field value is initially 0, and it is set to 1 by the network to indicate congestion. The sender starts by sending one packet to the receiver. The receiver acknowledges each received packet. The honest receiver's acknowledgment reflects the value of the congestion bit in the packet it received. Dishonest receivers are free to do anything. In response to receiving an acknowledgment that does not indicate congestion, the receiver sends two new packets. If congestion is indicated, the sender does not send any new data. In addition, the sender has a timer that fires periodically. If no packet has been sent since the last firing, the sender sends a new packet, to keep the information flow going.

We specify the network setup as having two nodes, one sender and one receiver, and we tell our tool to investigate manipulation by receivers. We also specify that protocol behavior should be measured in terms of the number of timer events. This measure indirectly captures the total time needed to send a particular number of packets. Because we do not explicitly simulate time, we cannot directly measure throughput. Finally, we emulate different network conditions by configuring different probabilities of setting the congestion bit.

The graphs in Figure 1 show the protocol behavior as a function of the network conditions. They show this behavior with both the honest receiver as well as with different cheating strategies. The set of cheating strategies in this case involve setting the congestion bit to 0, 1, or randomly. We conduct 200 trials for each combination of receiver strategy and network conditions, with each trial simulating the sending of 300 packets. We show all data points thus obtained, to demonstrate the variance with network conditions.

The figure shows how our tool can automatically determine which strategies work and quantify their impact. It shows that a misbehaving receiver can speed transfers by always setting the bit to zero and it can slow transfers by always setting the bit to one. Further, the misbehaving receiver can speed transfers even by setting the congestion bit randomly.

The graphs show that the impact of a cheating strategy is visible only under certain network conditions. For instance, the impact of setting the congestion bit to zero does not show until 40% of the packets have the congestion bit set and that of setting the bit randomly does not show until 60% of the packets have the congestion bit set. This behavior points to the importance of simulating different network conditions along with cheating strategies. Without simulating different network conditions, a gullibility checker might incorrectly infer that certain strategies are unsuccessful.

7. RELATED WORK

We are directly inspired by the recent success [7, 17, 22, 11] of systematic search techniques, as implemented by a model checker, in finding safety and liveness errors in system implementations. This class of research focuses on scaling to large systems and on nondeterminism arising from the timing of various events in the system. The de-

monic nondeterminism considered is fairly restricted and is limited to the network dropping packets or a system reboot occurring at an arbitrary instant. A straightforward extension of these techniques to include actions of an all-powerful malicious attacker does not scale.

Another related class of prior research applies formal verification techniques to validate abstract models of security protocols against malicious attacks [13, 15, 20, 18, 5]. Our work is most closely related to the use of model checking [20, 5] to systematically enumerate the behavior of the protocol in the presence of an attacker aiming to acquire a secret only known to honest participants. The attacker models considered are powerful and can generate arbitrary packets by combining parts of previously exchanged messages and publicly known encryption keys. However, due to the inherent state-space explosion problem, the problem instances considered are small hand-abstracted models of security protocols with very few message exchanges. In contrast, our work targets larger general-purpose network protocol implementations. We also check more complex properties, in particular those that compare the attacker against an honest version under the same network conditions. The price for these generalizations is that our approach is appropriate for finding attacks but not for producing a formal proof of correctness.

Automated fuzzing techniques [16, 2], which subject a system to random sequences of inputs, can find many errors in systems, some of which can lead to malicious attacks. As an extension, directed random testing [9, 3, 8] generates inputs based on symbolic execution of the program. These techniques have been used to find implementation errors, particularly errors in validating inputs. Our focus is on finding vulnerabilities in the protocol design itself. Therefore we restrict ourselves to well-formed input packets. However, as mentioned earlier we are interested in adapting fuzzing and testing techniques to our setting, in order to reduce the search space.

8. CONCLUSIONS

We have proposed and defined the problem of automatically checking protocols for gullibility, i.e., their vulnerability to manipulation by some of the participants. We identified the challenges in building a practical tool for this task and proposed several techniques to address them. We are currently developing a prototype checker using these techniques. Early results from using our tool to check the ECN protocol are promising, automatically identifying vulnerabilities and the network conditions under which they are exploitable. We are excited to improve our tool and apply it to new classes of protocols.

Acknowledgments We thank Stefan Savage, Amin Vahdat, and David Wetherall for feedback on this paper. This work was supported in part by NSF Grants CCF-0427202, CCF-0545850, and CNS-0725354.

9. REFERENCES

[1] A. Barbir, S. Murphy, and Y. Yang. Generic threats to routing protocols. Technical Report RFC-4593, IETF, Oct. 2006.

[2] Browser Fuzzing for Fun and Profit. http://blog.metasploit.com/2006/03/browser-fuzzing-for-fun-and-profit.html.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[4] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *OSDI*, Dec. 2002.

[5] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with brutus. *ACM Trans. Softw. Eng. Methodol.*, 9(4):443–487, 2000.

[6] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *The 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[7] P. Godefroid. Model checking for programming languages using verisoft. In *Principles of Programming Languages (POPL)*, Jan. 1997.

[8] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[10] C. Killian, J. W. Anderson, R. Baud, R. Jhala, and A. Vahdat. Mace: Language support for building distributed systems. In *PLDI*, June 2007.

[11] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in system code. In *NSDI*, Apr. 2007.

[12] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.

[13] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996.

[14] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Encouraging cooperation in multi-hop wireless networks. In *NSDI*, May 2005.

[15] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[16] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[17] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI*, Dec. 2002.

[18] L. C. Paulson. Proving properties of security protocols by induction. In *CSFW '97: Proceedings of the 10th IEEE workshop on Computer Security Foundations*, page 70, Washington, DC, USA, 1997. IEEE Computer Society.

[19] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *Computer Communication Review*, 29(5), 1999.

[20] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *CSFW*, pages 106–115, 1998.

[21] D. Wetherall, D. Ely, N. Spring, S. Savage, and T. Anderson. Robust congestion signaling. In *ICNP*, Nov. 2001.

[22] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *OSDI*, pages 273–288, 2004.