Finding Protocol Manipulation Attacks

Nupur Kothari
USC

Ratul Mahajan
Microsoft Research

Todd Millstein
UCLA

Ramesh Govindan
USC

Madanlal Musuvathi
Microsoft Research

ABSTRACT

We develop a method to help discover manipulation attacks in protocol implementations. In these attacks, adversaries induce honest nodes to exhibit undesirable behaviors by misrepresenting their intent or network conditions. Our method is based on a novel combination of static analysis with symbolic execution and dynamic analysis with concrete execution. The former finds code paths that are likely vulnerable, and the latter emulates adversarial actions that lead to effective attacks. Our method is precise (i.e., no false positives) and we show that it scales to complex protocol implementations. We apply it to four diverse protocols, including TCP, the 802.11 MAC, ECN, and SCTP, and show that it is able to find all manipulation attacks that have been previously reported for these protocols. We also find a previously unreported attack for SCTP. This attack is a variant of a TCP attack but must be mounted differently in SCTP because of subtle semantic differences between the two protocols.

Categories and Subject Descriptors

C.2.2 [Computer communication networks]: Network protocols–protocol verification

General Terms

Security, Reliability

1. INTRODUCTION

That network protocols can be vulnerable to attacks is well-known. By sending unexpected or malformed messages that exploit bugs or inadequate defenses (e.g., buffer overflows) in protocol implementations, adversaries can crash or hijack victims. Over time, research has led to a good understanding of such attacks and developed general tools and techniques to detect and mitigate them [4, 16, 20, 22]. Examples include automatically generating vulnerability signatures to filter malicious inputs and tracking the influence of tainted inputs on critical segments of the code.

In this paper, we consider a different and a more subtle class of attacks that we call *manipulation attacks*. In these attacks, the goal of the adversaries is not to crash or hijack the honest participants but to induce other behaviors that benefit the adversaries or harm the honest participants. Such attacks have been identified in several common protocols. Savage et al. found three different ways (e.g., by acknowledging packets even before they are received) in which an adversarial TCP receiver can manipulate the sender into sending at a rate faster than that dictated by congestion control dynamics [24]. Ely et al. showed that an ECN (Explicit Congestion Notification) receiver can manipulate the sender into ignoring congestion by simply flipping a bit in the packet headers [11]. Manipulation attacks that starve the honest participants have been identified for the 802.11 MAC protocol as well [1].

Instead of relying (only) on implementation bugs in protocols, manipulation attacks leverage the fact that individual participants do not have complete knowledge of network conditions (e.g., degree of congestion) or other participants' intent. Adversaries can exploit this incomplete knowledge by misrepresenting network conditions or their own intent in their messages to honest participants and can thus induce undesirable behavior in them. The induced behavior may be valid in certain circumstances but incorrect under current circumstances; for instance, increasing the sending rate is desirable when there is no congestion but undesirable otherwise. Often, adversaries may need to repeat such manipulations several times in order to achieve their goals such as performance gains for themselves or starving the honest participants.

Our long-term goal is to make protocols robust to manipulation attacks. In this paper, we focus on developing a method to help developers (and researchers) (a) identify whether a protocol implementation is susceptible to manipulation attacks, and (b) compute the sequence of messages that can be used to mount the attack. To our knowledge, these capabilities do not exist today; all manipulation attacks mentioned above were discovered through manual inspection of protocols by researchers. Techniques developed for other kinds of protocol vulnerabilities are unsuitable for detecting manipulation attacks. For instance, vulnerability signatures for detecting manipulating messages are hard to develop because such messages can be valid under certain circumstances.

Discovering manipulation attacks is challenging. Whether a single packet is successful in manipulating the honest participant towards a goal (e.g., increasing connection throughput) depends upon the goal itself, the network topology and traffic, and the internal protocol state (e.g., connection state, number of outstanding packets) at the honest participant. A brute force search of the entire space of packets, goals, topologies, traffic, and protocol states is clearly intractable. Even worse, successful attacks may require repeated manipulations where each input packet must leave honest participants in a valid state to continue execution, yet steer them towards the adversary's goal. This requirement significantly increases the search space.

To reduce the search space, we assume that developers specify goals as well as topologies and traffic. We posit that developers can provide these inputs based on their expertise and intuition. For instance, specifying the goal involves identifying a resource that may be vulnerable to manipulation (e.g., bandwidth) and protocol actions at honest participants related to that resource (e.g., sending a packet). Developers can iteratively explore many possible goals with varying topologies and traffic.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'11, August 15–19, 2011, Toronto, Ontario, Canada.
Copyright 2011 ACM 978-1-4503-0797-0/11/08 ...$10.00.

```
1   void ack_received(char *buffer){
2           int num_pkts_to_send;
3     if(num_pkts_in_flight == 0)
4       return;
5     num_pkts_in_flight--;
6     num_pkts_sent++;
7     int ecn_bit = ((packet *)buffer)->ecn_bit;
8
9     //if ecn bit is zero, send two
10    //packets in response else send none
11    if (ecn_bit == 0)
12      num_pkts_to_send = 2;
13    else
14      num_pkts_to_send = 0;
15    if(num_pkts_to_send > 0)
16      send_pkts();
17  }
```

(a) ECN Sender

```
1   void pkt_received(char *buffer)
2     int ecn_bit = ((packet *)buffer)->ecn_bit;
3     packet *ack_buffer;
4           num_pkts_rcvd++;
5     ack_buffer = create_ack(buffer);
6
7     //if the ecnbit is non-zero,
8     //set it in the ack header
9     if (ecn_bit != 0) {
10      ack_buffer->ecn_bit = ecn_bit;
11    }
12
13    //send the ACK to the sender
14    send_ack(ack_buffer,
15      ((packet *)buffer)->src);
16  }
```

(b) ECN Receiver

Figure 1: Code for a simplified version of the ECN protocol

Even with these inputs from developers, the space of possible manipulation attacks, defined by the space of possible messages and internal protocol state, is too large to allow for an exhaustive, blind search. To explore this space in a scalable and precise manner, we leverage two complementary program-analysis techniques.

The first is symbolic execution [15], a general method for static analysis of code, which we use to infer conditions on an incoming message and on the current state that may lead an honest participant to execute actions consistent with the adversaries' goals.

Because static analysis of complex code is inevitably imprecise, we combine it with a form of concrete, system-level testing. We run the protocol implementation and, based on information from symbolic execution, modify messages such that the honest participant is repeatedly induced to exhibit the identified behaviors. By comparing protocol runs with and without modifications, developers can determine if manipulation attacks can succeed. Additionally, the sequence of modifications provides information on how the attack was mounted and can guide the development of fixes.

While other researchers have used one or both of these techniques to find software bugs [6, 13, 28, 30], current approaches aim to identify a single bad input. We combine these techniques in a qualitatively different way, using symbolic execution to approximate single inputs that can manipulate the honest participant in its current state, and then using concrete execution to search for a sequence of inputs that constitutes an effective attack. Further, we adapt these techniques, sometimes using domain-specific optimizations and appropriately managing the trade-off between precision and scalability, to be able to handle full-fledged implementations of complex protocols such as TCP and SCTP [25].

We implement our method in a tool called MAX (Manipulation Attack eXplorer) and demonstrate its efficacy by applying it to implementations of TCP, the 802.11 MAC, ECN, and SCTP. For the first three protocols, we show that MAX is able to find and quantify the impact of all the manipulation attacks that researchers have discovered manually. We show that SCTP too is vulnerable to a manipulation attack, a finding that we believe has not been previously reported in the literature. While this attack is similar to one of the attacks on TCP, the way it is mounted in SCTP is different because of different receiver window semantics in the two protocols. MAX's analysis recognizes these semantic differences and infers the correct manipulation method for each protocol. We also show how MAX can not only find manipulation attacks but can also help developers gain confidence that their implementations are robust to certain attacks or that their protective measures are effective.

While these results are promising, our work represents only a first step towards a general tool for finding manipulation attacks. In the future, we plan to investigate the generality of MAX and its techniques by applying it to other, less well-explored protocols. Such an investigation will also help understand the prevalence of vulnerabilities to manipulation attacks in protocol implementations.

2. DEFINING MANIPULATION ATTACKS

Consider a protocol with two or more participants. The honest participants correctly follow the protocol, but adversarial participants may deviate arbitrarily. A manipulation attack has the following two characteristics:

1. Adversaries induce the honest participants to change behavior in a way that benefits adversaries or harms the honest participants.

2. The induced behavior is actually protocol-compliant under some circumstances and network conditions.

These characteristics rule out attacks where, for example, a rogue wireless node unilaterally jams the medium or where crashes are induced in honest participants. These attacks are important as well but are not the focus of our work; techniques exist to detect and mitigate them (e.g., [4, 6, 20, 22]).

Because they are subtle and indirect, most manipulation attacks have a third characteristic:

3. A single manipulative act is insufficient to mount an effective attack. Instead, the manipulation must be repeated many times in order to have a noticeable impact.

We use the ECN protocol as a running example in this paper, to illustrate a simple manipulation attack and the conceptual ideas behind our approach. ECN enables the network to notify the senders of congestion. When a packet encounters congestion, the congested router sets the ECN bit in the header of the packet. The receiver copies the ECN bit into the acknowledgment packet that it sends out. The sender becomes aware of the congestion when it receives an acknowledgment with the ECN bit set, and it can react by reducing its sending rate.

Figure 1 shows snippets of code for a simplified version of ECN that we wrote for illustrative purposes. The sender (Figure 1a), on receiving an acknowledgment from the receiver, checks the ECN bit in the header. If it is set, the sender does not send out any new

packets. If the ECN bit is not set, which indicates that there is no congestion in the network, the sender sends two new packets to the receiver. The receiver (Figure 1b), on getting a packet from the sender, obediently reflects the ECN bit from the packet header into the acknowledgment packet that it sends back to the sender.

Ely *et al.* [11] report a manipulation attack on ECN in which the receiver is adversarial, while the sender is honest. The adversarial receiver never sets the ECN bit, irrespective of what it receives from the routers, thus manipulating the sender into believing that the network is not congested (even if it is). As a result, the sender sends two new packets upon receiving each acknowledgment. This behavior results in increased throughput for the receiver at the expense of other flows traversing the congested router.

The attack above exhibits all three characteristics of a manipulation attack. The adversarial ECN receiver manipulates the behavior of the honest sender for its own benefit. Further, this manipulated behavior is protocol-compliant under some circumstances, namely when there is no congestion. For this reason, the attack is hard for the ECN sender to detect. Finally, the manipulation must be repetitively performed to have a noticeable impact on throughput. Modifying a single message only causes two extra messages to be sent, which will not improve the receiver's throughput by much.

A number of manipulation attacks have been manually discovered in other widely used network protocols as well. For example, an adversarial TCP receiver can manipulate an honest sender into sending faster and obtain an unfair share of bandwidth, by sending duplicate acknowledgments (DupACK spoofing), by acknowledging individual bytes rather than whole packets (ACK division), or by acknowledging packets before they are received (Optimistic ACKs) [24]. 802.11 implementations have been found to be vulnerable to manipulation attacks that prevent honest nodes from gaining access to the channel and thus starve them, by setting large values in the `duration` field in packet headers [1].

3. FINDING MANIPULATION ATTACKS

Our goal is to enable systematic exploration of manipulation attacks in protocol implementations. Although analyzing abstract protocol specifications is easier, we analyze real implementations because inadvertent programming errors can produce vulnerable implementations even if specifications are robust. Further, specifications of complex protocols are typically imprecise and, because different implementations make different design choices, each may have a different set of potential vulnerabilities. For instance, not all manipulation attacks for TCP [24] were successful for all implementations. Analyzing implementations allows us to uncover both vulnerabilities in a protocol's specification and vulnerabilities created by specific implementation choices.

3.1 Challenges and Approach

A tool that only takes the protocol implementation as input, and automatically outputs all possible manipulation attacks on it, would need to search a very large space. This space includes all possible goals of manipulation (e.g., starving honest participants or inducing them to send more traffic), all possible network configurations (i.e., topology and traffic workload) as well as all possible combinations of message contents and internal protocol state (because the protocol behavior is a function of both these factors).

We reduce this search space in two ways. First, we require the tool's user (a developer or a researcher) to specify the manipulation goal. This involves specifying a protocol action (e.g., sending a packet, in the ECN example) whose repeated execution could lead to a manipulation attack. Second, we require the user to specify the network configurations under which the potential for a manipulation attack should be explored. While it may be possible to automate the search of goals and network configurations, we leave this to future work. Our approach leverages the intuition and understanding that developers have of the likely protocol actions that can be exploited and the likely network conditions under which manipulations can succeed.

With these inputs, we are left with the problem of exploring the space of possible packet inputs and protocol internal states. This problem poses two challenges. The primary one is scalability—the space of possible manipulations is huge. Adversaries interact with honest participants by sending packets, and there are in principle 2^{12000} possible ways of crafting a 1500-byte packet. Even if we focus only on protocol headers, there are 2^{160} possibilities for a 20-byte header. Further, the exact contents of a packet that successfully manipulates an honest participant may depend on the (dynamically varying) state of the participant.

This intractably large space of possibilities implies that blind exhaustive or random search is unlikely to be effective. We thus leverage program semantics to efficiently search the space. In particular, we use symbolic execution, a static program analysis technique, of the protocol implementation to determine the set of packets that may induce honest participants to execute actions of interest, as a function of their current state. This set is determined statically, without actually running the protocol, and is represented in a concise, symbolic form. Packets that are not in the set are not considered further in our approach, allowing us to dramatically reduce the search space for manipulation attacks.

The second challenge is precision. In principle, symbolic execution alone can identify manipulation attacks. In practice, however, static analysis of complex code can be imprecise. Real protocol implementations can contain tens of thousands of lines of code with many low-level constructs. For instance, the Linux implementation of TCP that we analyze is over 14K lines of C code with many pointers and loops that are hard for static analysis to handle precisely. Imprecision in the analysis can lead, for instance, to false positives (i.e., attack strategies that do not work in practice) that may render the tool unusable. Further, unlike crash attacks, being able to reach an action of interest once does not imply that an effective manipulation attack exists. Instead, the adversary may need to repeatedly drive honest participants to execute certain actions. That ability depends on complex, dynamic factors (e.g., network congestion) that are hard to capture statically. Finally, even if a manipulation can be successfully repeated, it may or may not have a significant enough impact compared to an honest run of the protocol to be considered a success. It is difficult for static analysis to quantify the impact of an attack.

We address the limitations of symbolic execution above by combining it with *adversarial concrete execution*. We run the protocol implementation and use information from symbolic execution about possible packet manipulations, along with information about the current execution state, to perform manipulations that induce honest participants to execute actions of interest repeatedly. This produces an end-to-end demonstration of manipulation attacks, a catalog of one or more sequences of packet manipulations that led to the attack, and a quantification of the impact of these attacks. Furthermore, concrete execution ensures that no false positives are reported. However, as we discuss later, there can be false negatives.

3.2 Method Overview

We now describe our method for systematically exploring manipulation attacks. We focus on its conceptual elements and leave to §4 the details of a tool that embodies our ideas.

We assume that one or more adversaries manipulate an honest participant, called the *target*, through their protocol messages. While we allow for multiple (possibly colluding) adversaries, we currently allow only one target and leave to future work extensions to handle multiple targets. There can be other (non-target) honest participants in the network.

We also assume that the attack is launched by repeated manipulations of individual messages. To our knowledge, this is true of all manipulation attacks that have been uncovered to date. In theory, however, more complicated manipulation attacks are possible. For instance, they may require a specific sequence of distinct manipulations or may require that a manipulation occur at a specific time. We leave the exploration of these variants to future work.

The main input that we require from the users of our tool is a set of *vulnerable statements* representing the actions of interest for achieving a manipulation goal. Our approach is agnostic to the specific actions represented by these statements. We expect that users would typically first identify a resource (e.g., memory, network bandwidth, etc.) whose vulnerability they are interested in exploring and then use code points where these resources are allocated as vulnerable statements. For example, if manipulation of memory allocation is of interest, statements that allocate memory would be deemed vulnerable. Similarly, for network bandwidth, statements where packets are transmitted or where variables that control the sending rate are updated would be deemed vulnerable. The vulnerable statements are by themselves benign and are likely executed frequently even in the absence of attacks. Thus, unlike assertion failures, their mere reachability does not signify the possibility of a successful attack.

We test if adversaries can repeatedly drive the target to these vulnerable statements and if that significantly impacts the protocol performance compared to scenarios without manipulation. To quantify the impact of an manipulation attack, users specify a set of metrics defined on the resource of interest. For a resource like network bandwidth, this may simply be the number of times the vulnerable statement (that transmits a packet) is executed. For a resource like memory, this may be the cumulative number of bytes allocated during the execution.

We accomplish our goal of finding manipulation attacks by using the two analysis techniques discussed earlier, in a way that exploits the strengths of each. Symbolic execution prunes the search space dramatically by providing a set of constraints on network messages and the target's state that must be satisfied to reach the vulnerable statements. Adversarial concrete execution exploits knowledge of the target's current state to repeatedly solve these constraints and observe if an effective manipulation attack can be mounted. We describe our use of these techniques in more detail below.

3.2.1 Path exploration using symbolic execution

Symbolic execution [15] simulates the execution of code using a symbolic value $\sigma_{\mathtt{x}}$ to represent the value of each variable `x`. As the symbolic executor runs, it updates the symbolic store that maintains information about program variables. For example, after the assignment `y = 2*x` the symbolic executor does not know the exact value of `y` but has learned that $\sigma_{\mathtt{y}} = 2\sigma_{\mathtt{x}}$. At branches, symbolic execution uses a constraint solver to determine the value of the guard expression, given the information in the symbolic store. The symbolic executor only explores the branch corresponding to the guard's value as returned by the constraint solver, ensuring that infeasible paths are ignored. If there is insufficient information to determine the guard's value, both branches are explored. In this way, a tree of possible program execution paths is produced. Each path is summarized by a *path condition* that is the conjunction of branch choices made to go down that path.

We use symbolic execution on the target's implementation to explore possible paths that lead from message reception to vulnerable statements. Consider the ECN sender code in Figure 1a and assume that statement `send_pkts()` at line 16 is the sole vulnerable statement. Of all the possible paths through this code, symbolic execution determines that there is only one feasible path that reaches this statement, with the following associated path condition:

$$\sigma_{\mathtt{num_pkts_in_flight}} \neq 0 \ \& \ \sigma_{\mathtt{ecn_bit}} = 0 \ \& \ \sigma_{\mathtt{num_pkts_to_send}} > 0$$

This path is the one that takes the `else` branch at line 3, the `then` branch at line 11, and the `then` branch at line 15. All other paths either can never lead to the vulnerable statement (e.g., the `then` branch at line 3) or are infeasible. For example, the `else` branch at line 11 sets `num_pkts_to_send` to 0, which prevents the `then` branch at line 15 from being taken.

We convert each path condition into an *input constraint*, a predicate on symbolic values (i.e., on the incoming protocol message and internal protocol state) that is sufficient for execution to go down that path. This conversion is straightforward given the symbolic store. For the path condition described above for the ECN example, the input constraint is:

$$\sigma_{\mathtt{num_pkts_in_flight}} \neq 0 \ \& \ \sigma_{\mathtt{buffer->ecn_bit}} = 0$$

3.2.2 Attack emulation using concrete execution

We use the input constraints derived above to emulate manipulation attacks using concrete execution of the protocol implementation on a network configuration specified by the user. During this execution, we intercept each protocol message from an adversary that is directed to the target. An *adversarial module* then attempts to modify the message so as to drive the target to execute a vulnerable statement. We use a constraint solver to find concrete values for the message fields that, given the current internal protocol state of the target, satisfy one of the input constraints.

Per the input constraint above for ECN, whenever the sender's internal state has a nonzero value for `num_pkts_in_flight`, setting `buffer->ecn_bit` to 0 will drive the sender to execute the vulnerable statement. The adversarial module performs this modification and passes it to the target's message handling function. The control is then transferred to the protocol implementation to process this message.

By intercepting every message from the adversary to the target and using the adversarial module to modify it, our concrete execution attempts to repeatedly drive execution to a vulnerable statement. The user can determine, using the output values of the specified impact metrics and by comparing with a non-adversarial run, whether the implementation is vulnerable to a manipulation attack.

4. MAX DESIGN AND IMPLEMENTATION

We now describe our tool, called MAX, which uses the approach above to find manipulation attacks in protocol implementations written in C. Program analysis in C is challenging since the language has many low-level constructs, but we choose it because it is widely used in real-world protocol implementations.

MAX consists of three main components (Figure 2). The *path explorer* uses static analysis to find feasible paths and constraints that lead to vulnerable statements. The *adversarial module generator* uses the results from the *path explorer* to generate an adversarial module, which mimics the behavior of an adversary attempting to launch a manipulation attack. The *attack emulator* runs the protocol implementation using the adversarial module.

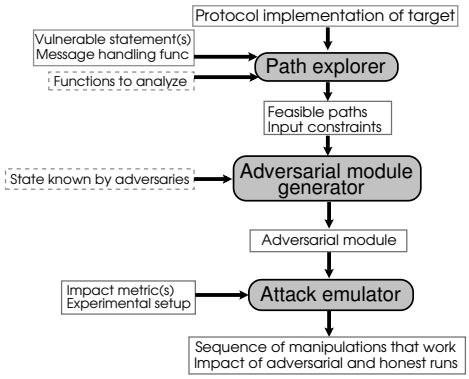


Figure 2: Overview of MAX. The dashed inputs are optional.

4.1 Path Explorer

The path explorer takes as input the protocol implementation of the target and along with the vulnerable statements and the function(s) that represents the starting point for handling incoming messages. Protocol implementations typically have a few, easily recognizable places where the processing of incoming messages begins (e.g., the `message_rcv()` function in our TCP implementation). An optional input is a set of functions in the protocol implementation that should be explored. This input can be used to focus the analysis on a subset of the code when the user is interested in (faster) exploration of specific aspects of the protocol. The default is to consider the entire implementation.

The path explorer operates in three steps. First, it parses the source code and build a control flow graph (CFG) for each function; nodes in a CFG are statements, and edges represent the order of execution of statements. From these individual CFGs, it builds a composite CFG, rooted at the protocol's message reception function, where there is an edge from a statement in one CFG to the root of another CFG if that statement calls the function corresponding to the target CFG. We use the CIL infrastructure for C program analysis [21] for this task. Second, to reduce the complexity of symbolic execution and improve scalability, the path explorer uses a standard reachability analysis to identify and remove nodes that do not reach vulnerable statements. Finally, it uses our symbolic execution engine (described below) on the pruned CFG. Whenever one of the vulnerable statements is reached during this execution, we convert the associated path condition into an input constraint. After we store this information, execution continues, until the entire CFG has been explored.

4.1.1 Symbolic execution engine

We implement our own symbolic execution engine rather than using an existing one [5, 30] in order to manage the tradeoff between precision and scalability, which is critical for handling real protocol implementations. Further, most existing engines work only on whole programs, while we wanted the flexibility of analyzing a subset of the implementation if desired by the user.

MAX's symbolic execution engine is implemented as an interprocedural analysis in CIL and uses the Z3 constraint solver [10]. It is multi-threaded, to take advantage of multi-core architectures.

As mentioned previously, symbolic execution converts program statements into relationships (or constraints) among symbolic values. Much of this processing is fairly standard except for some types of program constructs that we describe below.

Function calls: We first check if the body of the function being called is included in the user's set of functions to explore and if it is part of the pruned CFG for the implementation. If both conditions are true, the function needs to be analyzed. For such functions, the symbolic executor adds its formal parameters to the symbolic store, mapping them to the arguments specified in the function call, and starts traversing the function body.

If the function need not be analyzed, the symbolic executor assigns fresh symbolic values to variables that might be modified by the call and continues with the statement following the function call. This approach is conservative, essentially assuming that the variables can take on any value after the call, and it can cause the tool to treat some infeasible paths as possibly feasible. However, our attack emulation step weeds out any such false positives, thus maintaining precision.

Loops: A naive treatment of loops would require the symbolic execution engine to consider an unbounded number of paths, since the actual number of loop iterations cannot always be known statically. To address this problem, the symbolic execution engine partitions the possible paths through a loop into two cases: *i*) when the loop guard is false and the body is never entered; and *ii*) when the loop body in entered at least once [12]. We invoke Z3 (constraint solver) to evaluate the loop guard, and if the guard is known to be false, we consider only the first case. If the guard is known to be true, we consider only the second case. Otherwise, we consider both cases.

For the false case, the loop body is simply skipped and simulation continues at the subsequent statement. For the true case, the engine conservatively simulates the effect of an arbitrary number of iterations of the loop. It first invalidates information in the symbolic store about any variable that may be modified by the loop body, similar to what is done for calls to un-analyzed functions. Then, the loop body is symbolically executed once. Finally, the loop guard is assumed to be false, its negation is added to the path condition, and symbolic execution continues after the loop.

Low-level constructs: Network protocol implementations make heavy use of features such as pointer manipulation, type casting, and arrays. One option for handling such features uniformly is to model all variables as byte arrays [5]. While highly precise, this approach dramatically increases the complexity (and thus reduces scalability) of symbolic execution. Instead, we use a single symbolic value for most variables. For compound types like structs and arrays, we use a symbolic value per field and introduce these symbolic values lazily as each field is encountered.

In order to retain precision, we employ domain knowledge about how network protocols use low-level features in order to devise precise ways of simulating their execution. For example, consider a statement of the form `x = (struct_b *)y`, where `y` is of type `struct_a *`. In general, it is difficult to deduce the relationship between the two struct types, which is necessary to transfer any knowledge in the symbolic store about `y` and its fields to `x`. We leverage the fact that when these structs represent packet headers, typically one of the two structs is embedded as a field of the other. We thus search for such an embedding in the struct definitions. If found, it is straightforward to update the symbolic store with information about `x`. If we cannot determine the relationship between the two structs, we conservatively learn nothing about `x` and its fields.

We use a few other simple heuristics to improve the precision of symbolic execution in the presence of other low-level features. For example, when trying to resolve a function pointer, we only consider functions having the same signature as the function pointer, whose addresses were stored somewhere in the protocol code.

In the implementations that we study, we find that symbolic execution retains precision for 75-80% of the analyzed statements that update state. Because our analysis approach is conservative, most

of the imprecision leads to false positives (which are later eliminated). However, there is one source of imprecision that may lead to false negatives. We are sometimes unable to link header fields of an incoming message to the corresponding variables in the target. For instance, bits 33-64 in the TCP header correspond to the sequence number. The target may parse these bits to obtain the value of a variable called `seqno`. If the complexity of the parsing code prevents us from correctly determining how the value of `seqno` is instantiated, we would not be able to analyze the impact of an adversary manipulating the sequence number field. In our experiments, we find that this is not a significant limitation because for all attacks that we study, we are able to link the relevant header fields to internal variables. To accommodate implementations where this does not hold, our analysis can be extended to accept this linkage as an input.

4.1.2 Emulation-driven optimization

For highly complex protocols, the path explorer may need to explore an extremely large number of paths (on the order of billions). Inspired by symbolic execution engines that use a mix of symbolic and concrete execution [13], we handle this challenge by introducing a profiling step before path exploration. We run an instrumented version of the protocol implementation on the network configuration that is used during attack emulation (see below) and automatically record the set of internal states encountered at the target. We seed the path explorer with this information, which explores only paths consistent with at least one of these concrete states and deems other paths as infeasible. In this way, the path explorer focuses on the network conditions likely to be encountered during concrete execution, improving scalability without sacrificing precision.

4.2 Adversarial Module Generator

The adversarial module generator analyzes input constraints for the paths discovered by the path explorer to generate an adversarial module. As mentioned in §3, this module computes message contents that, when received by the target in it's current state, are likely to cause it to execute a vulnerable statement.

By default, we assume that the adversary knows the values of all internal state variables of the target and can use them for crafting messages that lead to vulnerable statements. This adversary model ensures that MAX can find manipulation attacks that might be possible with an omniscient adversary that can infer all state variables through inference based on protocol dynamics or out-of-band information. However, users can explore weaker adversary models by specifying the subset of state variables that are known and a subset of message fields that can be manipulated.

The adversarial module resides at the target, where it intercepts and modifies incoming messages from adversaries. It is compiled with the protocol implementation and includes a function for modifying messages. A call to this function is inserted in the protocol code where message handling begins, enabling interception. This function first reads the current (visible) state of the target and then invokes the Z3 constraint solver to find an assignment of values for message fields that satisfy an input constraint provided by the path explorer. The message is then overwritten by these values and returned. From the target's perspective, it appears that the adversary itself had crafted and sent this message.

Given input constraints for multiple feasible paths and multiple satisfying assignments for the same input constraint (and thus possibly multiple ways to modify the message), there are many ways to explore the space of adversarial executions. Our current strategy is quite simple. We consider input constraints in an arbitrary order and use the first satisfiable one. To reduce the time spent searching for a satisfiable constraint, we cache this satisfiable constraint and, for future messages, first check if it still satisfiable. We use the default solution returned by the solver for the satisfiable constraint, rather than explore all feasible solutions.

The strategy above can in theory lead MAX to miss some manipulation attacks (i.e., a false negative). For instance, we would miss an attack that relied on sending the target down paths P1, P2, P3 in that order if we emulated a different order. We would also miss attacks that required a different assignment of values than the one returned by Z3 for the same input constraint. In practice, however, we find that this simple strategy is quite effective (§5) at finding manipulation attacks (perhaps because such attacks do not require a high degree of sophistication). In the future, we will extend the adversarial module with more exhaustive search strategies of the type used by model checking tools [14, 19, 31].

We find that in some cases the attack goals are more conveniently expressed not as vulnerable statements, towards which emulation should be driven, but as "not-vulnerable" statements, which should be avoided (§5.5). For example, it may be useful to explore manipulations of TCP that work by avoiding statements that reduce the congestion window. We thus allow users to specify such statements. The path explorer treats them just as vulnerable statements and generates input constraints for them. However, the adversarial module solves for the negation of those constraints.

4.3 Attack Emulator

The goal of the attack emulator is to find a sequence of manipulations that constitute an effective manipulation attack. The inputs to the emulator include the impact metrics and network configuration. The user specifies impact metrics using variables in the target whose values capture the impact of the manipulation attack. If such variables do not already exist, users can instrument the code to introduce new variables. One instrumentation that we provide by default is a variable that counts the number of times vulnerable statements are executed. In many cases, this instrumentation measures the cumulative resource usage for the resource of interest (e.g., when bandwidth is the resource and `send_pkt()` is a vulnerable statement). In our current implementation, the user is required to manually configure the network topology and traffic to be used for emulation. It is, however, straightforward to automate this process by having users specify the desired topology and traffic in a configuration file, which we can use to configure the emulation experiment.

The attack emulator runs unmodified protocol implementations for all participants except the target. For the target, it uses the protocol implementation, compiled with the adversarial module, so that the incoming messages can be intercepted and modified.

The attack emulator performs two sets of runs: an adversarial run in which messages from adversaries to the target are modified and an honest one in which the messages are not modified. For complex protocols with many paths, interception of messages can introduce noticeable delay in message processing, which can slow protocol dynamics. For instance, for TCP this delay translates to participants observing higher round trip times and thus slower growth in sending rate. For a fair comparison between the two runs, the adversarial module is executed for the honest run as well, but without the modification step, to introduce comparable delays. For multi-party protocols, messages received by the target from non-adversaries are delayed similarly (during both adversarial and honest runs).

After completing the runs, the emulator outputs the impact metrics for both runs, a trace of the manipulations used by the adversarial module, their success in reaching the vulnerable statements, and symbolic constraints on the selected paths. The impact metrics let users determine whether the manipulation attack was effective. The

	LoC (K)	Paths explored (K)	Feasible paths (K)	Avg. path length (#conds)	Avg. compute time / path (s)
TCP	14.2	7,747	6,163	83	0.59
SCTP	12.5	2,124	359	134	10.11
802.11	11.0	7,288	237	63	15.26
ECN	7.6	2	1	28	0.58

Table 1: Complexity of the protocols that we analyze

other pieces of information can be used to infer the attack method. This process currently requires manual inspection of the constraints and manipulations. In theory, this inspection can be tedious. For the attacks that we find in §5, however, even though the attack emulator chooses multiple different paths to reach the vulnerable statements, the paths have similar constraints on the inputs (header fields) and protocol state variables. In such cases, manual inspection sufficed for inferring the header manipulations needed for mounting the attack. We emphasize that this inspection is not required to establish whether or not the manipulation attack was successful, but only to find the manipulations required to recreate the attack. Developing tools to help with the inference of attack strategy from emulator traces is an interesting avenue for future work.

5. EVALUATION

We use MAX to analyze implementations of four diverse protocols: TCP, 802.11, ECN, and SCTP. Before describing its efficacy in finding manipulation attacks in these protocols, we quantify their complexity and the performance of MAX when analyzing them.

5.1 Benchmarks

Table 1 shows different measures of protocol complexity. Each implementation has thousands of lines of C code. These lines represent the core protocol logic and do not include header files.

The statistics related to paths are based on running the path explorer for each of these protocols. While the symbolic execution ran to completion for ECN, for the other protocols, we ran it for two weeks before collecting these statistics. We used the emulation-driven optimization (§4.1.2) only for TCP. We see that for each protocol, over 2K distinct paths are explored, with over 7.7M paths being examined for TCP. Handling such complex code is challenging for any software analysis tool.

Scaling benefits of symbolic execution. Two measures shed light on the value of static analysis in our method. The first is the fraction paths that it can recognize as infeasible, so that they need not be analyzed further. Table 1 shows that this fraction ranges from 21-97%. This fraction is exceptionally low for the case of TCP (21%) because a large number of infeasible paths were pruned out before symbolic execution by using emulation-driven optimization.

Another measure stems from contrasting with a strategy proposed in prior work [29] for finding similar attacks without symbolic execution. This work proposes that, rather than exploring all possible bit patterns in the header, users restrict the search to header fields that likely impact the resource being manipulated, based on expert knowledge of the protocol. For example, in TCP, restricting the exploration to the sequence number field may suffice, which would yield 2^{32} possible inputs instead of $2^{40\times 8}$. Even if we ignore the error prone nature of manually identifying important fields, the value of symbolic execution is impressive. For TCP, we get 6M feasible paths, which is over three orders of magnitude lower than blindly manipulating the 32-bit sequence number. The scaling for SCTP is even better, where the number of feasible paths is $359K$, but the attacks we find below require manipulating two header fields with 32 bits each. Even if we assume that these two fields can be independently explored [29], this represents a scaling factor of over five orders of magnitude ($\frac{359K}{2\times 2^{32}}$).

The scaling benefits computed above are an underestimate because a packet's ability to manipulate and reach the vulnerable statement depends on not only the header values but also the internal state of the target. While a blind search would have to explore all possible internal states, MAX compactly represents each path predicate in terms of the header values and internal state needed to traverse the path. A single packet from this set, sent when the internal state has the appropriate values, suffices to exercise this path.

Computational complexity of symbolic execution. Table 1 also shows the average time taken to compute a feasible path for all protocols. This time ranges from 0.6 seconds for ECN and TCP, to 10-15 seconds for SCTP and 802.11. This disparity arises for different reasons for ECN and TCP. Firstly, ECN is a much simpler protocol, and the average number of conditions that need to be solved (path length in Table 1) is much lower than that for other protocols. In TCP, since a large number of infeasible paths have already been pruned by emulation-driven optimization, the number of paths that need to be explored before finding a feasible path is much lower. Using emulation-driven optimization for SCTP and 802.11 would have likely lowered their computation time as well.

5.2 Case Study I: TCP

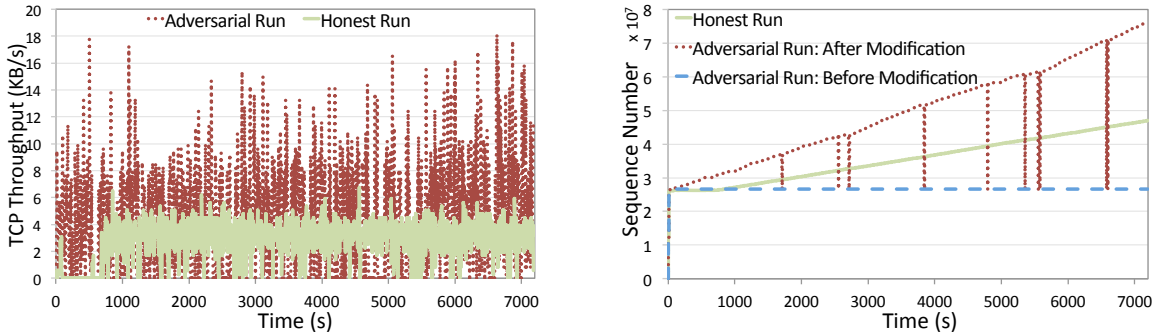
We now report on what MAX finds for each protocol, starting with TCP, which is one of the most complex network protocols and thus represents a good stress test. Our TCP implementation is Daytona [23], a user-space port from the Linux 2.3.29 kernel.[1] As shown above, it has 14K lines of code and over 7M paths. We wanted to check if, in this complex implementation, MAX can find attacks that have been previously discovered using manual inspection. We show that it is not only able to find vulnerabilities that exist but also helped confirm robustness to those that do not.

Finding the Optimistic ACK attack. In this attack, an adversarial TCP receiver sends ACKs with sequence numbers for packets that it is yet to receive. These Optimistic ACKs fool the sender into believing that more packets have been received and thus sending faster than with honest ACKs. If the traffic goes over a shared bottleneck, such an adversary would receive better performance at the expense of others. For this attack to work, however, the ACK numbers need to be within a range that represents the current window of unacknowledged sequence numbers.

In this experiment, we used a topology consisting of one target and one adversary communicating on the same LAN. We emulated a shared bottleneck by setting up background TCP traffic. The sender was deemed as the target. The relevant sender-side state variable that determines when new packets should be sent is the number of outstanding bytes (sent but not acknowledged). We specify as vulnerable statements places in code that decrease this variable, and thereby increase transmission opportunities. We ran the attack emulator for two hours and used the number of bytes transmitted (throughput) as the impact metric.

MAX successfully discovered the Optimistic ACK attack on the TCP sender by modifying the ACK sequence numbers. In particular, it automatically inferred the range of sequence numbers for which an ACK would reduce the number of outstanding packets. Even when a packet was dropped and a duplicate ACK was sent, MAX was able to automatically modify it such that its sequence

[1] We picked a user-space implementation for ease of experimentation. MAX can be applied to kernel-space protocols as well.



(a) Throughput **(b)** ACK Sequence Numbers

Figure 3: TCP behavior for the adversarial and honest runs for the Optimistic ACK attack.

number was in the queue of outstanding packets. MAX's modifications did not result in connection termination (§6).

Studying MAX's behavior in more detail, we see that it successfully manipulated the sender into executing the vulnerable statements for all but 22 times out of 11,480 packets. The 22 packets were received when the number of outstanding packets was zero, so MAX (correctly) found no feasible path that led to the vulnerable statement. This ability to determine if modifications would lead to vulnerable statements and compute the correct modifications when they can exemplifies the precision of MAX's analysis techniques.

Figures 3a depicts the impact of the attack by plotting the throughput for the adversarial and honest runs of the attack emulator. The average throughput in the honest run is 2.9 KBps. Manipulation increases it to 4.1 KBps, for a gain of over 40%.

Figure 3b illustrates the progression of the ACK sequence numbers. The middle curve shows a slow increase for the honest run. The other two curves show the sequence numbers before and after modification by the adversarial module. We see that modifications causes a faster growth than the honest run, which fools the sender into sending more packets. The sharp decrease in the sequence number in certain places is for the ACKs that were not modified.

Confirming robustness to attacks. Two other manipulation attacks have been reported earlier for TCP. These involve inducing the sender into quickly increasing the congestion window by acknowledging individual bytes instead of whole packets and by sending duplicate ACKs [24]. Fooling the sender into increasing the congestion window will give the receiver more throughput. These attacks do not work for all TCP implementations.

To determine if our implementation is vulnerable, we conducted an experiment similar to the Optimistic ACK attack above except that we specified statements that increase the congestion window as vulnerable. In this case, MAX was unable to repeatedly force the sender into executing the vulnerable statements. Sporadic manipulations of the congestion window did occur but did not lead to a noticeable increase in the adversary's throughput.

Manual inspection confirmed that the Daytona implementation indeed has safeguards [24] against these attacks. Because it is highly error-prone, manual inspection by itself is a weak indicator that an implementation is robust to certain attacks. The more systematic exploration of MAX provides a stronger safeguard.

5.3 Case Study II: SCTP

SCTP is a new transport protocol designed for reliable delivery and targets multi-homed environments [25]. To our knowledge, no manipulation attacks have been reported for SCTP [27]. We show that SCTP is vulnerable to attacks that are similar to those in TCP, though the exact mechanics of the attacks differ.

SCTP supports multiple independent streams within a connection and ensures sequenced, reliable delivery with each stream. This makes SCTP headers different (and richer) that those of TCP and some of the fields have different semantics. For instance, SCTP has two sequence numbers, the stream sequence number (SSN) used for ordering messages in a stream, and the transmission sequence number (TSN) used for reliable delivery. Given such differences, it was unclear to us if SCTP had vulnerabilities similar to TCP.

In these experiments, we used a complete user-level implementation of SCTP [26], with over 12K lines of code. The setup was similar to the one for the Optimistic ACK attack in TCP. That is, we deemed as vulnerable statements that decrease the number of outstanding bytes.

During attack emulation, MAX manipulated the SCTP sender into executing the vulnerable statement by modifying the cumulative TSN ACK field in all ACKs received at the target. This field represents the largest TSN of a packet received at the receiver, such that all packets before it have been successfully received. MAX increased the value of this field such that it was always less than or equal to the TSN for the last packet sent by the target (Figure 4a). This manipulated the SCTP sender into reducing the number of outstanding bytes and into sending faster than the honest run, for nearly 200s. (We explain the period after 200s below.) Within this time, this manipulation enabled the adversarial receiver to obtain almost 5 times more throughput than the honest receiver. (Figure 4b). Thus, for moderate sized transfers at least, we find that SCTP is susceptible to a manipulation attack in which the receiver optimistically increases the TSN ACK field. While such an attack has been reported previously for TCP, that SCTP is also vulnerable was not previously known.

However, for longer flows, SCTP's vulnerability to manipulating the number of outstanding bytes appears reduced, as shown in Figure 4b. Within a stream, when a data packet is lost (around 200s mark), subsequent packets are buffered at the receiver and the receiver window decreases significantly, preventing the sender from sending more packets. This loss is never recovered because MAX manipulates the TSN ACK to cover the lost sequence number and the situation persists. Because of the vulnerable statement that we use, which does not directly control the resource of network throughput and is not impacted by the receiver window, MAX does not infer that this window can limit network usage.

To evaluate if SCTP is vulnerable to longer-term Optimistic ACK manipulations, we experimented with specifying as vulnerable state-

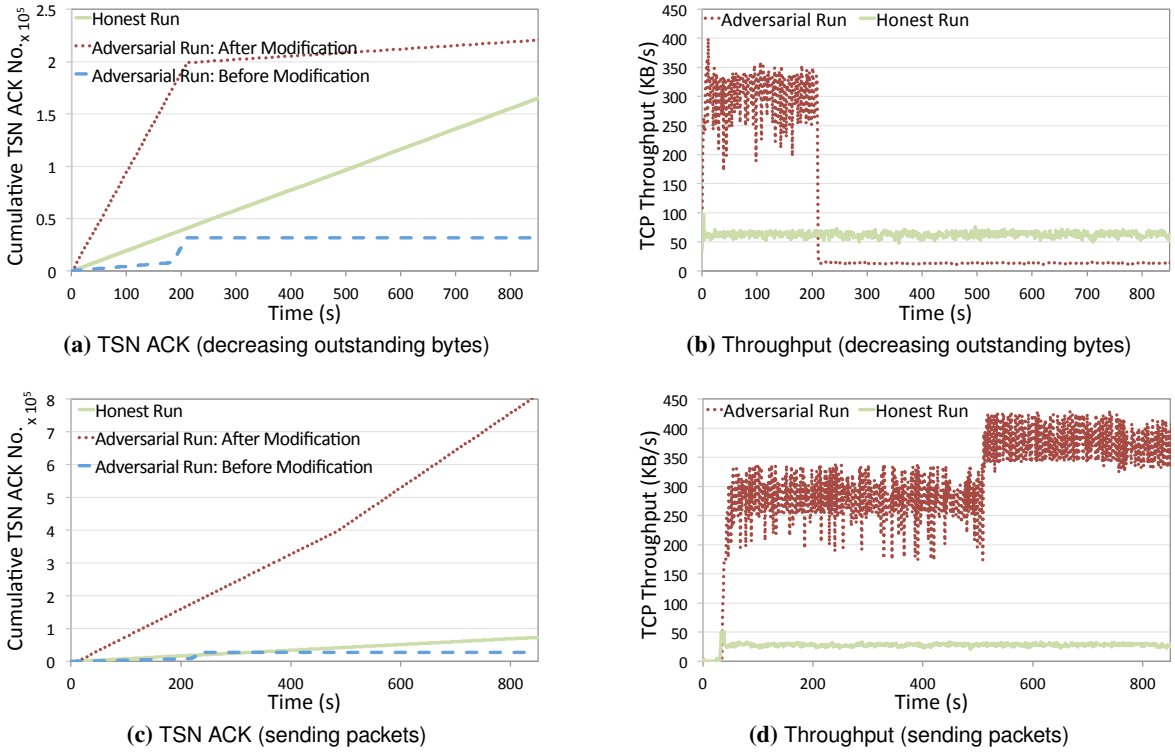


Figure 4: SCTP behavior with two vulnerable statements: decreasing outstanding bytes (top) and sending packets (bottom)

ments that send packets, which directly impacts network usage. We found that MAX now manipulates both the cumulative TSN ACK field as well as the receiver window, correctly inferring the values that lead to packet transmissions. Figures 4c and 4d show the results for the TSN ACK field and the throughput obtained. The impact of manipulating both fields is that even after a packet is dropped (around the 250s mark), the adversary is still able to obtain a higher throughput than the honest receiver, unlike in the previous experiment. This occurs because MAX increases the receiver window values as well, which are very low prior to the manipulation. Overall, the adversarial run achieves a 10x throughput over the honest case. (We do not fully understand the reasons behind the jump in throughput around the 500s mark, which was present in each of the many trials that we conducted to rule out experimental and measurement artifacts. This jump is consistent with the faster growth rate of the TSN ACK field around the same time. It may stem from constants in the SCTP implementation that trigger faster transmissions under certain conditions.)

Thus, SCTP is susceptible to the Optimistic ACK attack for long flows as well, but both the receiver window and cumulative ACK fields must be manipulated. This behavior differs from TCP, for which we confirmed that the receiver window modification is not required, because of a subtle difference in the receiver window semantics between the two protocols. In TCP, packets received outside the sequence number space demarcated by the advertised window are dropped. Thus, during the Optimistic ACK attack the receiver window is always open and the adversary does not need to manipulate it. In SCTP, however, receivers store any received packet and discard packets only when there is no available buffer space—this behavior is necessary to support multiple streams in a connection. The adversary cannot mount a sustained attack without also manipulating the receiver window.

Finally, MAX was able to highlight a different vulnerability in our SCTP implementation. This implementation does not check that the cumulative TSN ACK value is less than or equal to the largest TSN transmitted. In the absence of this check, if a buggy proxy or a man-in-the-middle attacker sends a high TSN ACK value, inconsistent state will occur between the sender and receiver. (The ACK should contain the correct verification tag [25] for the connection to be accepted by the sender.) There is no recovery when that happens and the connection will eventually terminate. The experiments above use a version of SCTP with this bug fixed.

5.4 Case Study III: 802.11 MAC

We next used MAX on the 802.11 MAC protocol, which differs significantly from TCP and SCTP. It is a link-layer protocol and is broadcast-based. As for TCP, manipulation attacks have been reported for the 802.11 MAC [1]. By deeming as vulnerable statements that impact how a node uses air-time, the primary resource in 802.11, we show that MAX succeeds in finding these attacks.

We used the 802.11 MAC implementation in the Qualnet simulator. Qualnet is implemented in C++, but the core of the 802.11 code is written using the C idiom. We were able to analyze this code after removing bindings to the Qualnet scheduler and redefining a few C++-specific data structures. Our ability to focus program analysis on a relevant subset was a key enabler. Symbolic execution engines that perform whole program analysis would not have been able to do a similar analysis of the 802.11 code.

Finding the NAV attack. In this attack the adversary sets an abnormally high value in the duration field of the 802.11 header. The receiving nodes use this field to compute the Network Allocation

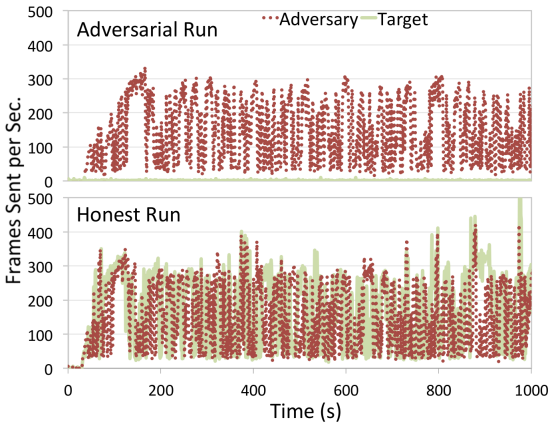


Figure 5: Impact of the NAV attack in the 802.11 MAC.

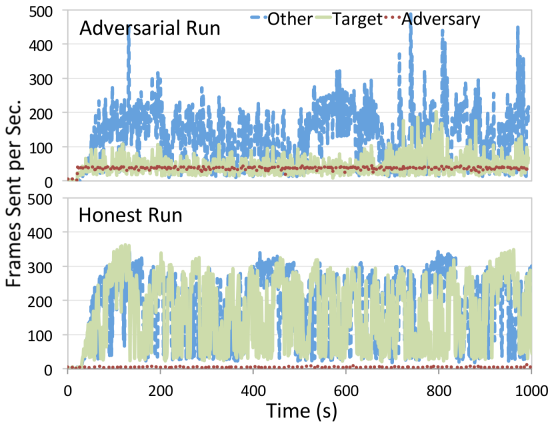


Figure 6: Impact of the RTS attack in the 802.11 MAC.

Vector (NAV) which dictates how long they should consider the medium as busy before they can transmit they own packets. High duration field values can starve honest participants and enable adversaries to gain an unfair share of the medium.

In this experiment, we considered the statement that sets the state of the 802.11 protocol to `WF_NAV` (*i.e.*,, waiting for the Network Allocation Vector value to expire) as vulnerable. A node cannot transmit any data while the NAV value has not expired, so forcing the NAV value to not expire can potentially induce a DoS attack.

We used as network configuration a simple topology with one access point (AP), one target client, and one adversarial client, with each client sending an infinite demand flow to the AP. We also included five other honest participants, to see if the use of MAX in a broadcast setting creates side-effects that hinder its abilities (§6). We used the number of frames transmitted as the impact metric and allowed the modification of all header fields except for `FrameType`. Modifying this field leads to another attack (see below).

We found that MAX successfully finds the NAV attack by repeatedly setting an abnormally high value in the `duration` field of the packet sent by the adversary, resulting in the the target using this high value to compute its NAV. Figure 5 shows the impact of this attack. It plots the frames sent by the nodes in the adversarial and honest runs. We see a dramatic difference; the target is shut out almost completely in the adversarial run.

Finding the RTS attack. In this attack, the adversary sends an RTS frame to the target with a high `duration` field value. The target responds with a CTS frame with a high `duration` value. Any node that hears either the RTS or the CTS frame updates its NAV value and stays silent for a long time. The target too stays silent, awaiting a (long) data to arrive. Thus, by sending a single packet, the adversary can impact multiple participants, some of them not even within its broadcast range. This attack is akin to virtual jamming, since the sender effectively prevents others from using the channel without sending at a high rate itself.

In this experiment, we considered statements that set a sleep timer as vulnerable. By sleeping, the target is deprived of bandwidth, which makes it vulnerable to the attack above. We used an ad-hoc network, which enables nodes to transmit to each other without an AP, with eight nodes: one adversary, one target, and six other honest participants. There was a flow from the adversary to the target that transmits a low rate of one packet per second. We also instantiated an infinite demand flow from the target to an honest participant and another between two honest participants. We allowed modification of all fields in the header.

We found that MAX can successfully discover the RTS attack. The frames sent by the adversary are modified in two ways: the `FrameType` is set to RTS and the `duration` is set to a large value. Figure 6 shows the impact of this attack. It plots the frame rate the adversary (configured to be low), the target, and another honest participant. We see that the adversary succeeds in reducing the rate for both of the other participants, with a bigger impact on the target.

During analysis, we also found an interesting bug in Qualnet. In a few places in the code, there is no verification for sleep timer values being non-negative. Therefore, the input constraints that MAX creates for paths to those timer settings do not require non-negative values. During emulation, MAX sometimes chose a negative value, which caused Qualnet to crash. Thus, MAX's techniques may also help find implementation bugs that cause crashes.

5.5 Case Study IV: ECN

While ECN is logically a rather simple protocol, it is unique from an implementation perspective. Bits relevant to the protocol are carried in both the network layer (IP) and the transport layers (TCP). We show that this does not prevent MAX from finding the manipulation attack in ECN that we mentioned earlier (§2). We also illustrate the utility of MAX as a design tool by patching the implementation using a known fix and then using MAX to check that the patched implementation is indeed robust. Due to space constraints, we omit detailed impact graphs in this section.

Finding the manipulation attack. We used the ECN-enabled TCP implementation in Qualnet. Because the value of the received ECN bit directly determines if the TCP congestion window should be reduced, we deem as "not vulnerable" (§4.2) any statement that reduces the congestion window. Intuitively, an adversary can increase its throughput by continually forcing the target to avoid reducing the congestion window.

MAX correctly deduces that in order to avoid reducing the congestion window, the adversary should not reflect the ECN bit in its ACKs when the bit is set.

Confirming the robustness of the patch. We modified the Qualnet ECN implementation to protect against this attack using nonces [11]. The sender inserts a randomly-generated nonce in its packets, and congested routers delete the nonce to indicate congestion. When the sender receives an ACK, it reduces the congestion window if the nonce is deleted or differs from the original. To fool the sender that there was no congestion along a congested path, the adversarial receiver needs to guess the deleted nonce correctly, which is difficult to do often enough to substantially alter protocol behavior.

We used MAX on this modified version of ECN, after hiding the nonce state at the target to emulate an adversary that does not know the nonce values. The path explorer found the same code paths as before. However, during emulation, MAX was unable to reliably force execution down these paths because it could not reconstruct the nonce value. The honest and adversarial runs yielded similar throughputs. Hence the attack was not successful, and MAX was able confirm that the fix and its implementation are effective.

6. DISCUSSION AND FUTURE WORK

Our work represents a first effort to build a practical tool for finding manipulation attacks. It opens avenues for future investigation towards improving MAX and mitigating such attacks.

Extending MAX. The current design of MAX has several limitations that we plan to address in the future. These include extending it to handle multiple targets, incorporating a more exhaustive search over feasible paths, and allowing an adversary to send multiple messages or no message at all when an honest participant would have sent a single message.

Another limitation involves protocol implementations where the vulnerable actions are not executed immediately after receiving a manipulated message, but at a later point in time. For instance, imagine a protocol where manipulated messages update a data structure at the target, and later when a timer fires, this data structure is read and the vulnerable statement is executed. Such a programming paradigm makes it hard to link an incoming message to vulnerable actions. We are exploring if such behaviors can be handled by using taint analysis to infer that link.

In MAX, we place the adversarial module at the target, which has two advantages that we plan to exploit in the future. This placement can assess the worst-case effects of an attack because it gives access to the target's exact internal state. It can also accommodate multiple adversaries, allowing us to explore collusion between adversaries.

However, two mismatches occur due to this placement. First, it does not naturally handle link-layer broadcast protocols, in which a manipulation message from an adversary is visible to other participants. Our approach instead (logically) allows an adversary to send a manipulated message to the target while hiding the manipulation from others. Second, it introduces a possible mismatch between the protocol state at the target and the adversaries. The target assumes that the modified messages come from the adversary and updates its state accordingly, when in reality the adversary's implementation never sent the message. In our experiments thus far, we have not experienced significant side effects from these mismatches (even though we analyze broadcast-based 802.11 MAC), but we plan to explore methods that place the adversarial module at the adversary.

Finally, we plan to explore mechanisms that can reduce the amount of input required from the user. For instance, we may be able to use program analysis to automatically identify vulnerable statements from high-level descriptions of resources (e.g., throughput, memory, etc.); and we may be able to find most manipulation attacks by running the protocol over a set of common network topology and traffic configurations.

Mitigating Manipulation Attacks. While this paper focuses on finding manipulation attacks, our eventual goal is to make protocols robust to these attacks. We conjecture, however, that it is impossible to completely prevent all manipulations unless we can enforce correct protocol execution by all participants (e.g., using trusted hardware). The main reason is not code complexity (as for buffer overflow vulnerabilities) but partial information—each participant has a partial view of the state of the network (e.g., presence of congestion) or of other participants (e.g., if it received a packet). If all participants had complete knowledge, they would not need to exchange any information in the protocol headers. The need for information exchange opens the door to manipulation. (The role of MAX is to help identify which information can be exploited and how.)

While we cannot prevent all manipulations, we can certainly mitigate their impact. One approach is to verify received information and take protective action if discrepancies are detected. Sometimes, verification can be done by simply retaining historical information. For instance, ACK division in TCP can be detected by remembering where packet boundaries occurred in the byte stream. In protocols with more than two participants, verification can be done by comparing information from different participants (e.g., OSPF nodes check if link-state reports from the two ends of a link are consistent). It may also be possible to verify information by observing a participant's future behavior. For instance, NAV manipulations in 802.11 MAC can be detected post-facto by observing if nodes actually occupy the medium for the claimed duration.

Verification may be infeasible in some cases, however (e.g., verifying if a participant received a packet). Here, a second approach that relies on explicitly considering adversaries' incentives can help. In particular, we can modify the protocol exchange such that adversaries can only hurt themselves by manipulation. Examples are the use of nonces in ECN or anonymous packets in Catch to prevent wireless relays from dropping packets [17].

7. RELATED WORK

While we adapt and combine symbolic and concrete execution in a novel way, we build on work that uses these general techniques to detect other kinds of software vulnerabilities. The inspiration for MAX comes from the work of Stanojevic *et al.* [29], who defined the notion of protocol *gullibility* that is similar to, but broader than, our notion of manipulation attacks. They also present a preliminary tool to detect protocol gullibility based on concrete execution alone. To reduce the search space they require the user to input a set of attack strategies (e.g., only manipulate certain header fields). This tool could find the gullibility in a simplified version of the ECN protocol. In contrast, by using symbolic execution as well, we do not require that users input attack strategies and can also scale to complex protocol implementations.

Other relevant prior work can be divided into three categories. The first category uses static analysis alone to protect against (other types of) network attacks. For example, SAFER [7] uses taint analysis to identify inputs that can cause DoS attacks; and Elcano [4] uses symbolic execution to generate signatures that filter out inputs known to exploit protocol vulnerabilities, given an initial exploit that exposes the vulnerability.

To eliminate or reduce false positives inherent in this first category, the second category combines static analysis with concrete test generation [13, 28, 30]. For example, EXE [6] generates "inputs of death" that cause a program to crash, Brumley *et al.* present a technique to automatically generate exploits from software patches that target input validation vulnerabilities [3], and Bouncer [9] uses a combination of static and dynamic analysis to create filters that block bad inputs during program execution. These approaches aim to identify a single bad input. However, a single input is insufficient to mount an effective manipulation attack, which requires the ability to repeatedly induce honest participants into performing certain actions. Further, for manipulation attacks, a "bad" input can be perfectly valid under different conditions, which renders input filtering invalid. MAX thus first uses symbolic execution to approximate single inputs that can manipulate the target in a given state and then uses adversarial concrete execution to search for a sequence of inputs that constitutes an effective attack.

The third category uses concrete execution alone and is exemplified by several model checking systems. For example, CMC [18, 19] identifies generic errors and invariant violations in protocol implementations, and MaceMC detects violations of liveness properties for protocols [14]. More recently, CrystalBall [31] adapted model checking to an online setting by check-pointing a running system and using the checkpoints as the starting point for the model checker. Unlike the first two categories, model checking can find errors triggered by a sequence of events. However, model checking aims to identify errors in a setting where participants run the protocol implementation faithfully. In manipulation attacks adversaries can arbitrarily deviate from the protocol implementation in order to induce undesirable behavior. This possibility leads to a dramatically larger search space which makes a concrete execution-only approach intractable. MAX uses information from symbolic execution to help steer its adversarial concrete execution towards code paths that are likely lead to manipulation attacks.

Finally, Bethea *et al.* [2] recently described an approach and associated tool to perform online verification that a sequence of inputs received at the server could have been generated by an honest client. Like MAX, their tool combines the results of symbolic execution with state information derived from concrete execution. However, their tool cannot be used to detect manipulation attacks, because the sequence of inputs generated by an adversarial client may be perfectly valid under the right network conditions.

8. CONCLUSIONS

We presented a method and a tool to find manipulation attacks in which adversaries induce honest participants into undesirable behaviors. A novel combination of symbolic and concrete execution, and their adaptation to network protocol code, allows our tool to scalably and precisely analyze complex protocol implementations and find a diverse set of attacks.

We believe this combination is useful beyond finding manipulation attacks and opens the door to other forms of semantic analysis of protocols. For instance, a similar approach may be able to analyze how state flows across a network, with symbolic execution inferring what parts of a node's internal state are carried in outgoing messages and concrete execution observing how messages flow between nodes. The resulting analysis could enable, among other things, an automatic quantification of protocol complexity [8].

Acknowledgements

We thank the SIGCOMM '11 reviewers and our shepherd, Brad Karp, for feedback on this paper. This work is supported by the National Science Foundation award CNS-0725354.

References

[1] J. Bellardo and S. Savage. 802.11 denial-of-service attacks: real vulnerabilities and practical solutions. In *USENIX Security*, 2003.

[2] D. Bethea, R. Cochran, and M. K. Reiter. Server-side Verification of Client Behavior in Online Games. In *NDSS*, 2010.

[3] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy*, 2008.

[4] J. Caballero, Z. Liang, P. Poosankam, and D. Song. Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration. In *RAID*, 2009.

[5] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[6] C. Cadar, P. Twohey, V. Ganesh, and D. Engler. Exe: A system for automatically generating inputs of death using symbolic execution. In *CCS*, 2006.

[7] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *IEEE Computer Security Foundations Symp.*, 2009.

[8] B.-G. Chun, S. Ratnasamy, and E. Kohler. Netcomplex: A complexity metric for networked system designs. In *NSDI*, 2008.

[9] M. Costa, M. C. L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP*, 2007.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

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

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

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

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

[16] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *SIGCOMM CCR*, 32(3), 2002.

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

[18] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *NSDI*, 2004.

[19] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A programatic approach to model checking real code. In *OSDI*, 2002.

[20] V. Navda, A. Bohra, and S. Ganguly. Using channel hopping to increase 802.11 resilience to jamming attacks. In *IEEE Infocom Mini Symp.*, 2007.

[21] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CCC*, 2002.

[22] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[23] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum. Daytona: A user-level TCP stack. http://nms.lcs.mit.edu/~kandula/data/daytona.pdf, 2003.

[24] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *SIGCOMM CCR*, 29(5), 1999.

[25] Stream control transmission protocol. IETF RFC 2960, 2000.

[26] Stream control transmission protocol (SCTP). http://sctp.de/sctp.html.

[27] Security attacks found against the stream control transmission protocol (SCTP) and current countermeasures. IETF RFC 5062, 2007.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5), 2005.

[29] M. Stanojevic, R. Mahajan, T. Millstein, and M. Musuvathi. Can you fool me? towards automatically checking protocol gullibility. In *HotNets*, 2008.

[30] N. Tillmann and J. de Halleux. Pex: White box test generation for .NET. In *Tests and Proofs*, LNCS. 2008.

[31] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. Crystalball: predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, 2009.