Encouraging Cooperation in Multi-Hop Wireless Networks

Ratul Mahajan Maya Rodrig David Wetherall John Zahorjan
University of Washington

Abstract

We present Catch, a protocol that encourages cooperation in multi-hop wireless networks comprised of autonomous nodes. While these nodes depend on each other to relay packets, scarce bandwidth and energy resources may motivate some nodes to cheat and avoid packet forwarding. Catch employs a novel technique based on anonymous messages to significantly increase the difficulty of cheating with impunity. Existing routing protocols simply assume that nodes will act cooperatively and have no mechanisms to encourage this behavior or punish cheaters. Catch imposes minimal requirements and overheads on the system, and so is broadly applicable across routing protocols and system workloads. We show that Catch makes cheating counter-productive, thus encouraging the cooperative behavior on which multi-hop wireless networks are predicated. We evaluate Catch on an 802.11 wireless testbed, as well as through simulation.

1 Introduction

Experience has shown that when systems rely on the cooperation of autonomous nodes to achieve a communal goal, some fraction of the nodes will "cheat" by consuming global resources without faithfully carrying out their obligations to contribute. For example, measurements of peer-to-peer file sharing systems show that many users retrieve files without serving any [1, 33]. Cheating has also destroyed the popularity of commercial online games [31] and has been a longstanding point of contention for network congestion control [17, 18]. Approaches to preventing cheating have been of considerable interest in these arenas (for example, see references for peer-to-peer systems [23, 30, 13, 36], congestion control [34, 21, 5, 24, 26], and online games [7]).

This paper addresses cheating in an emerging cooperative environment – multi-hop wireless networks. There has been a dramatic increase in the deployment of 802.11 networks over the past few years, with a growing boom in WiFi hotspots and community networks [16]. Multi-hop networks, in which nodes relay packets for each other, are a natural next step. In infrastructure rich areas, relaying packets can reduce dead spots, lower power consumption [29], and increase network capacity [20]. Additionally, multi-hop wireless networks can be deployed in infrastructure poor environments more readily and at lower expense than traditional wireless networks. This is particularly important in rural or developing areas, but is also valuable in buildings without appropriate cabling infrastructure. Research examples of multi-hop networks include MIT's Roofnet [3], Microsoft's MUP [2], the Digital Gangetic Plains Project [9], and UCAN [25].

Cheating in a multi-hop network means the failure to forward packets for other nodes. Most existing designs for these networks ignore this issue, and in fact may even reward cheaters by routing traffic around them on the presumption that they have no useful connectivity to the rest of the network. This has two negative effects. First, all packet forwarding is concentrated through the cooperative nodes, decreasing both individual and system bandwidth and increasing their energy loads. Second, cheaters may partition what would otherwise be a connected network. Figure 1 illustrates this second effect in our 15 node testbed 802.11 system.[1] Each line in the graph shows the probability that the cooperative nodes are partitioned given varying numbers of cheating nodes (that forward no packets) and minimal acceptable link quality (where only links with at least the given delivery rate are considered useful). Figure 1 shows that even a few cheaters are capable of severing high quality connectivity, and that rampant cheating will partition the network even if the cooperative nodes try to use low quality links.

Prior work on preventing cheating in wireless networks has concentrated on the use of incentives or pricing schemes. However, we believe that each of the solutions of which we are aware is unsuitable for use in practice. Typically, they have employed some form of currency, using it to force nodes to forward packets in return for the right to have their own packets forwarded. There are two major impediments to putting such schemes into practice. First, they impose strong requirements on the infrastructure, such as centralized clearance services [40, 32] or trusted hardware [12]. Second, they are structurally bound to some notion of fairness, typically that the number of packet forwarded for and by a node can differ by only a small amount. This works well only when there are uniform traffic rates among all node pairs [35]. Our challenge is to determine how to discourage cheating without such restrictions or limitations.

[1]Our testbed infrastructure is described in more detail in Section 5.

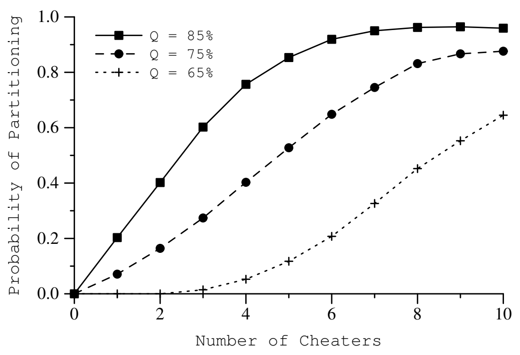

Figure 1: *Probability that non-cheating nodes are partitioned versus varying numbers of (randomly chosen) cheating nodes. Only links with delivery rates at least Q are considered useful.* (15 nodes total in the testbed.)

Our goal is a protocol to discourage cheating using only lightweight, unrestrictive, and generally applicable mechanisms. By doing so, our technique can be easily integrated with the rapidly evolving research results on multi-hop wireless networking that solve other problems. Our approach is to sufficiently "raise the bar" on cheaters, catching and punishing them quickly enough that their own self-interest encourages cooperation rather than cheating. Our solution is a protocol called Catch. Catch differs from earlier work in that it is statistical: it does not prevent a cheater from dropping packets, but elevates the risk of doing so enormously by detecting such behavior with high probability and punishing it. Additionally, the larger the percentage of packets dropped by a cheater the quicker the detection, thus limiting the benefit of any attempt to exploit the inherent statistical noise. In return for the weaker (statistical) guarantees that Catch provides, it is much more widely applicable than existing approaches.

We have implemented and evaluated Catch on an in-building 802.11b testbed. This setting includes the complex link quality factors that affect actual wireless systems, such as the unpredictable relationship between distance and loss rate, loss rates that shift over time, and asymmetric connectivities (where A can hear B, but B cannot hear A). Such factors significantly complicate the implementation of robust decision mechanisms where nodes monitor the behavior of their neighbors. These aspects of the problem have received little attention in earlier work, which is primarily based on simulation. We find that Catch is able to detect attempts at cheating by individual nodes both quickly and with high accuracy. The overhead for our scheme is modest, a nominal load of roughly 24Kbps per node in our testbed. With Catch there is no space overhead or cryptographic operations per data packet.

The rest of this paper is organized as follows. We describe our problem setting in Section 2. Section 3 presents our approach, which is based on the use of anonymous messages to estimate true reception rates and infer wireless connectivity to a cheating node. Section 4 describes Catch, our protocol for discouraging cheating. Section 5 describes our 802.11 testbed implementation, and presents the results of experiments that measure the detection accuracy, timeliness, and overhead of Catch. We analyze Catch across a range of parameters through the use of simulation in Section 6. Finally, Section 7 presents related work, and we conclude in Section 8.

2 Problem

Our goal is to deter cheating so that all nodes in the network cooperate and forward packets for each other. We make three assumptions. First, we assume that most nodes are cooperative (in the sense that they run a protocol we define) and cheaters are uncommon. This seems fitting for the problem domain since a reasonable level of cooperation must exist to form a network in the first place. We do not consider collusion amongst cheaters. Second, we assume omni-directional radio transmitters and antennas, so that nodes can overhear nearby communications. This matches common 802.11 usage today. Third, we assume that nodes have a long-lived identity that they cannot change at will. Such a notion of identity is required by most incentive-based schemes in both wireless and other domains. For current 802.11 hardware, MAC addresses do not provide such an identity. But mechanisms such as WPA in the upcoming 802.11i standard (already implemented by many vendors) should be sufficient for this purpose.

Note that we do not make assumptions regarding the routing protocol, traffic workload, power management scheme, or objective of the nodes (such as bandwidth maximization or energy conservation). This is because we wish our solution to work largely unchanged across any of these variables.

Existing protocols do not solve this problem. There are two simple ways in which a selfish node can cheat, one at the forwarding level and one at the routing level. The first way a node can cheat is by dropping all the packets it receives for forwarding without any further processing. Earlier work has suggested the use of a watchdog that leverages the broadcast property of wireless transmissions to detect this problem, coupled with the avoidance of these misbehaving nodes [27]. However, while this prevents packets from being forwarded to misbehaving nodes, it also means that the cheater has succeeded in its goal of offloading work to others.

The second way for a node to cheat is to simply refuse to send routing messages that acknowledge connectivity to itself via more than one other node. Because of asymmetry, other nodes that can hear the cheater cannot be sure that the cheater can hear them. The cheater will then appear to be a "dead-end," and no traffic will be sent to it for forwarding. We call this cheat *link concealment*. It is broadly applicable and, to the best of our knowledge, no existing routing protocols or policing schemes defeat it because the true connectivity of a node is known only by the node itself.

3 The Use of Anonymity

In this section we describe the key elements of our approach. At a high-level, we encourage cooperation by detecting and punishing nodes that cheat. One component of detection is the watchdog technique [27] mentioned earlier, which relies on the inherent broadcast nature of wireless networks. When one node sends a packet to another for forwarding, it *listens* to the wireless medium hoping to overhear whether the packet is in fact forwarded. The fraction of packets for which forwarding is observed is one of the inputs eventually used to decide if the neighbor is cheating. If so, all the neighbors of a cheating node punish it by *isolation*: all refuse to forward its packets for some period, effectively disconnecting it from the network.

This approach requires us to tackle two key problems:

1. Distinguishing a neighboring node that is cheating, i.e., dropping packets deliberately, from one that is not receiving them due to real wireless transmission errors.

2. Communicating that a node is cheating to all of its neighbors so that they can collectively punish it, even when the only path to those neighbors is through the cheater.

Each of these problems is difficult or impossible to solve perfectly in the general case. For the first, only a given node knows which packets it received and which it did not – this information is not externally observable. For the second, the cheater can simply refuse to forward messages it thinks are incriminating.

One insight of this paper is that anonymous messages, in which the receiver cannot determine the identity of the sender, can be combined with the threat of isolation to address both problems. Anonymous messages can be provided with current hardware by scrubbing the source addresses on packets.[2] This forces would-be cheaters to engage in sophisticated games with signal strength measurements if they are to infer the sender. For now, we assume that anonymity can be provided. We return to the impact of physical layer hints in Section 5.

[2]The source MAC address can be scrubbed for most current 802.11 cards.

The next two subsections describe how we tackle each problem. For consistency with the Catch protocol description in Section 4 we refer to a node that is being checked for cheating as a *testee* and its neighbors which are assumed to operate correctly as *testers*.

3.1 Anonymous Challenge Messages

The *anonymous challenge message* (ACM) sub-protocol allows a tester to estimate its true connectivity with the testee. This then allows it to determine if its data packets are being forwarded in good faith or deliberately dropped. To understand the protocol observe that for a selfish testee to stay connected at least one of its testers must be willing to forward its packets. Call this other tester the gateway.

In the ACM protocol, each tester regularly but unpredictably sends an anonymous challenge message to the testee for it to rebroadcast. The tester can check whether the testee does so because radio transmissions are broadcast and can be overheard by neighbors with reasonable success [27]. Assume that the tester will refuse to forward packets for the testee if it fails to overhear the rebroadcasts (since it believes the testee either has no connectivity or is cheating). Now, to preserve connectivity with the gateway, the selfish testee must rebroadcast challenges sent by the gateway. But since all challenges are anonymous, it cannot select only those of the gateway, and must respond to all of them. This in turn allows the other testers to determine that they have connectivity to the testee. The protocol is difficult to undermine even with weak anonymity because the likelihood of correctly handling a series of challenges decreases exponentially over time.

To use this protocol in practice, we must allow for wireless losses. Once we do so, the testee may drop some or all anonymous challenge messages. However, it cannot selectively inflate the loss rate on just some links because the challenges are anonymous. To make the addition of uniform loss counterproductive, testers look for gross discrepancies in either direction between the anonymous challenge loss rate and the estimated data packet loss rate. That is, if cheaters add loss to all challenges then they must also drop and retransmit their own data packets to avoid detection, a losing proposition that marginalizes the value of cheating in this manner.

3.2 Anonymous Neighbor Verification

The ACM protocol is not a complete solution because a cheater may handle anonymous challenges correctly but discard the data packets sent by some of its testers. While these testers will refuse to relay packets for the cheater, this is to no avail as long as the cheater has some willing gateway. To prevent this, all testers must collectively isolate a cheater. The *anonymous neighbor verifi-*

cation (ANV) sub-protocol allows testers to do so by exchanging information with each other via the testee, even though the testee may prefer to discard incriminating information.

ANV operates in two phases. In the first ("ANV Open") phase, the testers anonymously advertise their existence to each other. They send the cryptographic hash of a token that the testee is required to rebroadcast. As before, anonymous messages are used to prevent the testee from selectively excluding testers. If the testee does not rebroadcast these messages, the testers assume that it does not have connectivity or is cheating and do not relay packets for it.

In the second ("ANV Close") phase, each tester releases their token and reveals their identity as long as the testee has forwarded its packets correctly, according to the ACM protocol. Assuming that it is not feasible to invert the hash, a token can only be released by the tester who encrypted it. If each tester does not eventually hear all of the tokens it expects then it concludes that some other tester has incriminating information. This causes the cheater to be isolated by all testers. Thus, the testee must forward packets for all testers, because any of them can withhold its token. Note that it is crucial to the protocol that failure of the testee be signaled in the ANV2 step through the *absence* of a message. This prevents a cheater from interfering with such signaling, as it would be able to if we used a more straightforward positive signaling mechanism.

We make two further observations. First, as before, dropping messages in the first phase to exclude particular testers and their data packets is unlikely to succeed because the likelihood of correctly matching anonymous messages to testers decreases exponentially over time. Second, interference in the second phase of protocol by the testee is clearly unproductive because it can only lead to its isolation.

4 The Catch Protocol

Catch builds on the anonymous techniques above, adapting them for use in real, wireless networks.

4.1 Overview

Catch operates as a sequence of protocol epochs run between a *testee* node and its neighbors, who act as *testers*. Figure 2 provides two illustrations of the per-epoch protocol steps, one when the testee is cooperating and the other when it is cheating.

Epochs are delimited by EpochStart packets broadcast by the testee. During the epoch testers send data packet for the testee to forward and estimate link loss rates using the ACM sub-protocol. Testee's behavior during an epoch is evaluated at the end of that epoch. This evaluation and that of a small number of preceding epochs is used by the testers to judge whether the testee is cheating. Should they conclude that it is cheating, they punish it for an interval during which they all refuse to forward its packets.

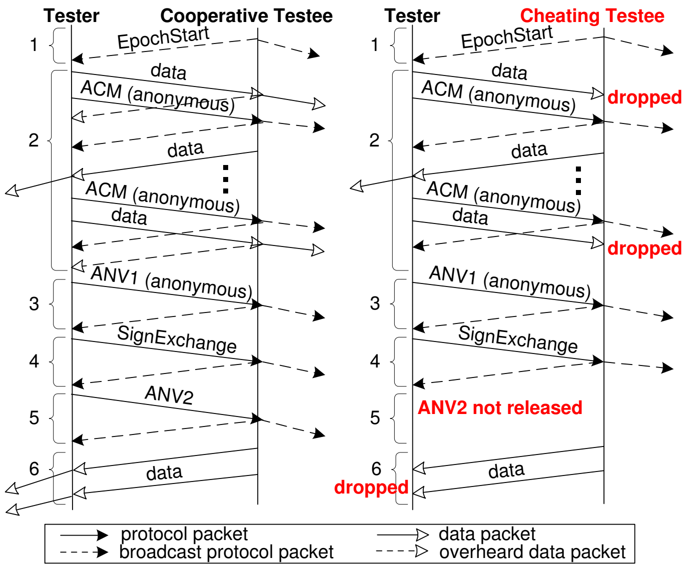

Figure 2: *Protocol flow. Packet exchange between a tester and a cooperative (left side) or cheating (right side) testee. Numbers on the left of the time sequence correspond to the protocol steps.*

1. *Epoch-Start.* The testee broadcasts an EpochStart packet that includes its identity and an epoch identifier. Nodes that receive this request participate as testers for this epoch.

2. *Packet Forwarding and Accounting.* During packet forwarding, testers use a watchdog [27], listening in promiscuous mode for forwarding transmissions of their packets, and count both the number of data packets sent and the number of forwarding transmissions overheard. During this period the testers also run the ACM sub-protocol (Section 3.1), sending anonymous challenges and counting their rebroadcasts.

3. *Anonymous Neighbor Verification Open (ANV1).* Each tester "opens" the two-phase ANV sub-protocol (Section 3.2) by sending an anonymous packet containing a hashed token to the testee for rebroadcast.

4. *Tester Information Exchange.* Each tester compares the fraction of its data packets it has heard the testee forward during the epoch to the fraction of its challenges it has heard reflected. It obtains a one-bit ("sign") result depending on which is greater: 0 for

challenges and 1 for data packets. It then sends its sign bit and identity to the testee for rebroadcast.

5. *Epoch Evaluation and ANV Close (ANV2).* Each tester determines whether the testee is operating correctly using its observations and the data from other testers. This is done with a pair of statistical tests described in the next subsection. If both tests pass and the testee correctly rebroadcast the one-bit result above, then the tester closes the ANV protocol by releasing its token.

6. *Isolation Decision.* An epoch fails for a tester if either of the tests above fail or it does not receive all of the tokens it expects. If too many epochs fail too quickly (Section 4.3) then the tester decides that the testee is cheating and punishes it by dropping its packets for a fixed number of epochs. By virtue of the protocol, all testers decide to punish a cheater at (nearly) the same time, so that it is isolated.

We increase the likelihood of all testers seeing all control packets (EpochStart, ANV1, SignExchange and ANV2) in two ways. First, we use retransmissions, e.g., if a tester does not hear the rebroadcast. Second, we use cumulative broadcasts, where the testee sends all of the information it has received on every transmission.

4.2 The Per-Epoch Tests

Each tester applies two statistical tests per epoch to determine whether a testee is behaving correctly. Each test is designed to be sensitive to distinct cheating strategy. The key challenge in both is to avoid mistaking volatile wireless conditions for misbehavior.

One cheating strategy for the testee is to drop packets from a particular tester in the hope that the consensus across neighbors will be that the cheater has passed the epoch, since all other testers should find its behavior acceptable. To detect this, each tester aggregates information about the testee's data packet forwarding and connectivity as estimated using anonymous challenges for the last three epochs. It then compares these two rates using the *z* test [28]. We found that quite high confidence levels (99% and higher), coupled with the hysteresis provided by using measurements from multiple epochs, provide a good balance between quick detection of cheating and a low rate of false accusations where cooperative nodes are accused of cheating.

The second strategy a cheating testee can employ is to uniformly drop some fraction of the packets received from each tester, making it hard for any one of them to conclude that cheating has taken place. To detect this, we employ the sign test [28] using the one-bit results exchanged by all testers. The sign test is based on the idea that if the testee is not cheating, its perceived forwarding and connectivity rates should have identical means. Thus, random fluctuations in each epoch should yield about as many results in which one exceeds the other as the opposite. Each tester accumulates the one-bit results for all epochs in which it has participated, and applies the sign test to decide if the balance is reasonable.

4.3 The Isolation Decision

Isolation of a testee is decided by all testers in parallel. An epoch fails if any of the testers fail the above tests; each tester knows if it failed the tests and learns if another tester failed when an ANV2 token is withheld. We do not isolate after a single failed epoch because, while it could indicate cheating, it could as well indicate an anomalous condition such as changed wireless conditions or control packets that were lost despite retransmissions and rebroadcasts. To allow for this, we require three failed epochs to isolate a testee. We do not use three consecutive epochs as a test because this would allow a cheater to succeed for two-thirds of the time. Instead, we use a three state finite state automaton (FSA) that moves to the right when an epoch fails and the left when an epoch passes. If the FSA falls off the right edge, the testee is isolated.

4.4 Protocol Fail-safes

Because Catch is designed to operate when some nodes act in an adversarial manner, we are as concerned about what happens when the protocol is not followed as when it is. In Appendix A we provide a short analysis by message type that shows cheaters cannot undermine the protocol in the absence of collusion.

5 Experimental Evaluation

This section describes our experiments with Catch on an 802.11b testbed. The purpose of this exercise is to test Catch under realistic wireless characteristics, such as the complicated packet loss behavior these systems exhibit [22].

5.1 The Testbed

In this section, we describe our testbed infrastructure and our implementation of Catch. The results of our experiments are presented in Section 5.2.

5.1.1 Physical Infrastructure

Our testbed is composed of 15 PCs running Linux 2.4.26. We use NetGear MA311 PCI network adapters (Prism 2.5 chipset), operating in the ad-hoc mode using the *hostap* driver. Each node also has a wired Ethernet interface to facilitate remote management of the experiments.

The testbed is located on a single floor of an office building, as shown in Figure 3. Apart from the usual sources

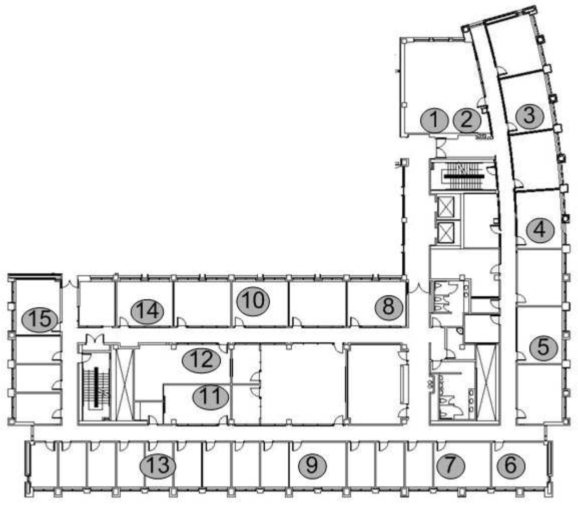

Figure 3: *Our wireless testbed, consisting of fifteen 802.11b nodes. The node locations are marked with circles.*

of noise and interference in an office environment, the building has its own dense deployment of wireless access points, including ten on the same floor as our testbed. These access points operate on 802.11b channels 1, 6 and 11. Our testbed operates on channel 1, thus competing with other wireless transmissions in the building. Such a setting, while noisy, is also realistic. The growing use of WiFi means that most 802.11 networks will be competing with each other in any public deployment [4].

We use static routing between nodes to factor out effects that stem mainly from the routing protocols. Wireless routing protocols are currently an open area of research [14].

Our system exhibits all the well-known real-world characteristics of wireless networks, including error rates that are not a simple function of distance, that are strongly asymmetric, and that vary widely over time. Figure 4 gives a static summary of these effects. It shows the average one-way delivery rate in each direction for each pair of nodes for which we observed a successful transmission in at least one direction. We computed the delivery rate by having each node broadcast 500 1000-byte packets over two minutes. The other nodes counted how many of those packets they received. The wide range of inter-node delivery rate shown in the figure, rather than a binary state of connectedness, has been previously observed for wireless environments by other researchers [4, 39].

5.1.2 Catch **Implementation**

We implemented Catch at user-level using the Linux *netfilter* framework to monitor and manipulate the packets sent, received, and forwarded by a node. The watchdog

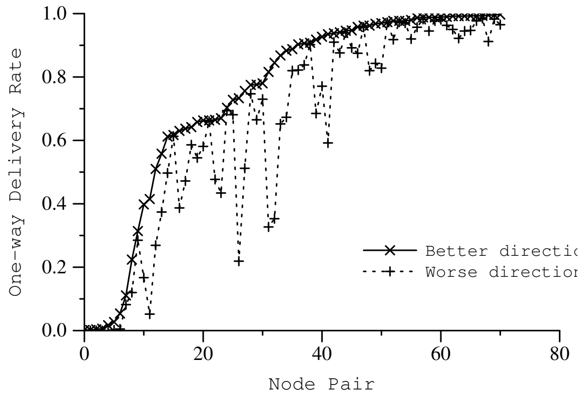

Figure 4: *For each node pair in the testbed, the fraction of sent packets successfully received in each direction. There are 105 pairs total in the testbed. Only node pairs with a non-zero delivery rate between them in at least one direction are shown.*

component of Catch also needs to overhear all packets sent by the node's neighbors regardless of their intended destination. To capture these packets, we operate our wireless network adapters in promiscuous mode and use the Linux pcap framework. The Catch protocol itself is written in ruby.

The watchdog mechanism needs to account for 802.11 MAC-level retransmissions while assessing the testee's data forwarding rate. Consider a tester judging whether the testee forwarded a particular data packet. The quality of the link between the testee and the recipient determines the number of retransmissions done by an honest testee, which in turn changes the probability that the tester will overhear this packet. To overcome this recipient-based variation, we measure the data forwarding rate using only the first transmission. A bit in the 802.11 MAC header enables us to distinguish original transmissions from retries. We mimic a similar behavior with ACM messages by broadcasting their responses only once.

We use the following parameters values for our evaluation, based on experiments across multiple settings. The length of an epoch is set to one minute. This provides a compromise between cheating detection speed and protocol overhead. There are fifteen anonymous ACM messages per epoch. This number is chosen to be sufficiently large for statistical decisions but small enough to add very little overhead. The confidence interval for the z test is 99.999%, and that for the sign test is 99.995%. Both experiments and simple analysis showed that very high confidence values are most effective. The size of ACM messages is 1500 bytes, which is the same as the MTU (maximum transmission unit) used by our network adapters. With smaller data packets such as TCP acknowledgements, the data loss rate can be less than that of the ACM messages. Our current implementation checks for the upper bound. We plan to explore tighter bounds using the

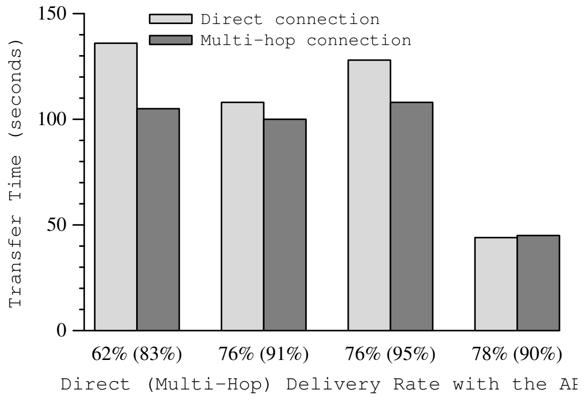

Figure 5: *Time to transfer 6MB from node 8 in Figure 3 by four other nodes via direct and multi-hop connections. The x-axis label gives the delivery rate of the direct connection with node 8, with the delivery rate for the multi-hop path in parentheses. The client nodes, from left to right, are 4, 6, 14, and 9.*

measured relationship between packet size and loss rate in the future.

5.1.3 Multi-Hop Performance

To convey a general sense of how our testbed behaves, as well as show that multi-hop routing is useful even when direct connectivity is available, we compared the performance of a single, centrally located access point (AP) setup to that of multi-hop routing. We simulate this condition by transferring a large file from one node, which acts as the AP, to four client nodes. In one set of experiments, those nodes communicate directly with the AP. In the other, they use multi-hop routes.

Figure 5 shows the results. We used node 8 (Figure 3) as the AP, and nodes 4, 6, 9, and 14 as the clients. Each client downloaded a 600KB file ten times. Each multi-hop route used a single intermediary node: 4:5:8, 6:7:8, 14:10:8, and 9:10:8. The x-axis labels in the figure give the delivery rate of the direct links, averaged over the two directions. The parenthesized numbers give an estimate of the quality of the two-hop path, computed as the product of the delivery rate of the individual links. In total, the use of multi-hop paths reduced download time by 16%, with per-node benefits ranging from 30% to -2%. The better performance of the multi-hop routes is due in part to the lower packet loss rates they enjoy. De Couto *et al.* have studied these issues in more detail [15, 14].

5.2 Catch Evaluation

In this section, we describe experiments that evaluate Catch for its speed and accuracy in detecting cheaters, its overhead, its impact on the throughput obtained by the cheating nodes, and the impact of subverting anonymity using signal strength.

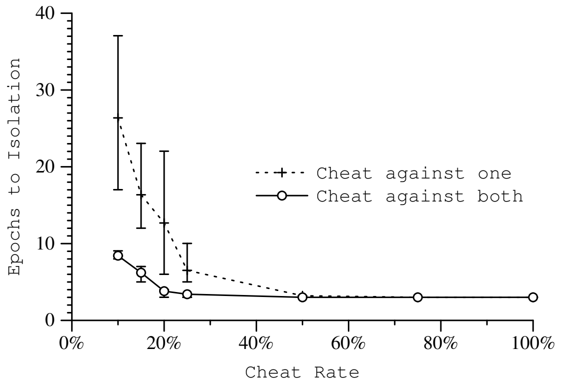

Figure 6: *The number of epochs required to detect cheaters in the testbed versus the fraction of packets a cheater failed to forward. Each point is the average of 10 experiments. Vertical bars represent the inter-quartile range.*

5.2.1 Detecting Cheaters

Our first experiment measures the speed with which Catch detects cheating. The more quickly a cheater is detected the less attractive cheating becomes.

To construct a base case for detecting cheating, we selected triplets of nodes such that both the first and the third node had a reasonable (>75%) delivery rate to the second node. The first and third nodes exchanged packets through the second node, which was configured to act as a cheater. The cheater randomly dropped a fraction of packets that it received for forwarding. We experimented with *cheating rates* of 10%, 15%, 20%, 25%, 50%, 75% and 100%. Cheating rates less than 100% mimic a situation in which the cheater tries to evade detection by appearing to be a cooperative but poorly connected node.

In our experiment, the first node acted as a client, and downloaded randomly selected files ranging from 1KB to 3MB in size from the third node. We started multiple download sessions in parallel so that even in the presence of a high cheat rate and TCP backoff dynamics, a minimum amount of traffic (roughly ten packets per epoch) is generated for the statistical tests.[3] Another download session was started soon after one of them finished.

The line "Cheat against both" in Figure 6 presents the results for the case when the cheater drops packets from both neighbors. It shows the average number of epochs required to detect (and so isolate) a cheater for varying cheat rates. Catch reacts quickly to cheating, and its reaction time decreases with cheat rate. Detection is almost immediate for very high cheat rates; recall from Section 4.3 that the minimum number of epochs that must fail before isolation is three. Even at the low cheat rate of 10%, Catch isolates the cheater in under 9 epochs on average.

[3]When the tester is not sending enough application packets, it can send real-looking dummy packets to examine the testee's behavior.

The curve labeled "Cheat against one" in Figure 6 shows the results for the case where the cheater dropped packets only for the client. We consider cheating against only one neighbor to evaluate whether a single victim can cause the cheater to be isolated. We chose the client as the victim because it sends smaller packets (download requests and TCP acknowledgements) and uses the loss rate of bigger ACM-probes to infer cheating. This is a challenging scenario for detection. We find that for high cheat rates the detection speed is just as fast as the "both" case. It is slower at lower cheat rates, but even at the low cheat rate of 10% the average detection time is less than 30 epochs. Thus, a cheater that persistently cheats even against one neighbor at a very low rate is eventually be caught and punished.

5.2.2 False Accusations

In this section, we check that the rapid detection of cheaters does not come at the cost of falsely accusing cooperative nodes of cheating, whether due to volatile wireless conditions or other factors.

We ran two five hour experiments in which all nodes were cooperative. Each node repeatedly downloaded files (from the same set as above) from randomly chosen servers, maintaining five simultaneous download sessions. This workload is high enough to saturate our network, stressing the accuracy of inference and increasing the probability of false accusations. We observed no false positives in the first experiment and a single false positive in the second. Node 8 incorrectly inferred that node 1 was cheating. It is difficult to measure the true rate of false accusations because they are rare events, but nevertheless we find this encouraging.

5.2.3 Coordinated Isolation

Detection is an effective deterrent in Catch only if all the well-connected neighbors of a cheater detect cheating at roughly the same time. If detection events are staggered, the cheater will remain connected to the network through a subset of its neighbors. In this section, we evaluate whether wireless conditions hinder the ability of the testers to reach a decision to isolate at roughly the same time.

In this experiment, we randomly selected 3 (20%) nodes as cheaters, who dropped all the packets they received for forwarding. All nodes executed a workload similar to the one in the previous section with the exception that cooperative nodes selected file servers only from the set of other cooperative nodes. We evaluated whether the non-cheaters were successful in isolating cheaters in a coordinated manner by observing the throughput obtained by the cheaters. Coordinated response implies that the throughput of the cheaters goes down to near zero.

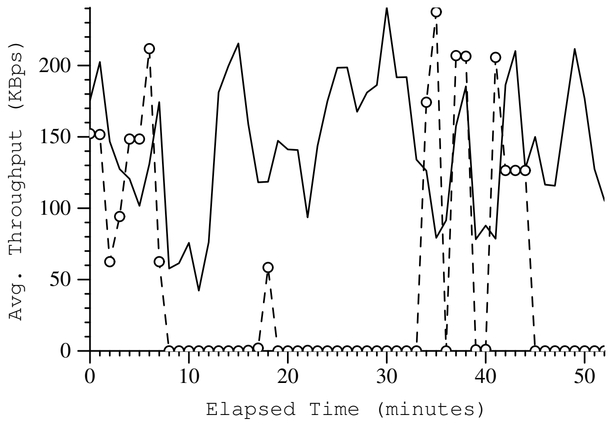

Figure 7: *Average throughput of cooperative nodes (solid line) and cheaters (dashed line) as a function of time. Throughput was calculated using one minute intervals. There were three cheaters. The punishment interval is 30 minutes.*

Figure 7 plots the average throughput obtained by the cheating and cooperative nodes. It shows that the cooperative nodes were successful in shutting out the cheaters. Roughly eight minutes into the experiment, all the cheaters were identified and isolated. The cheaters were allowed to send traffic again after the punishment interval of 30 minutes. The average throughput of the cheaters appears to recover before 30 minutes because different cheaters were isolated and released at different times.

5.2.4 Protocol Overhead

We report on the overhead of Catch in this section. We have made no attempt to optimize the protocol because its requirements are already modest.

Consider the activity for a pair of neighboring nodes in an epoch, both playing the role of tester and testee. The packet overhead of Catch comes from its messages which have different sizes and frequencies: StartEpoch (40 bytes), ACM challenges and responses (1500 bytes, 15 times per epoch), ANV open and close (100 bytes), and sign exchanges (40 bytes). These packets come to a total of 0.6 packets or 758 bytes per neighbor per second. Our testbed has less than four well-connected neighbors per node on average, which means that the protocol overhead is less than 2.4 packets or 24 Kb per second per node. This is roughly 0.2% of the 802.11b capacity (11 Mbps) and 0.04% of 802.11a/g capacity (54 Mbps).

We found the processor consumption of Catch to also be very reasonable. Informally observed using *top* during our experiments, it took at most 10% of the CPU on Pentium-IV 3 GHz nodes. Much of this is an artifact of our ruby, user-level implementation of Catch where each packet that passes through the local machine or is promiscuously overheard crosses the user-kernel boundary once

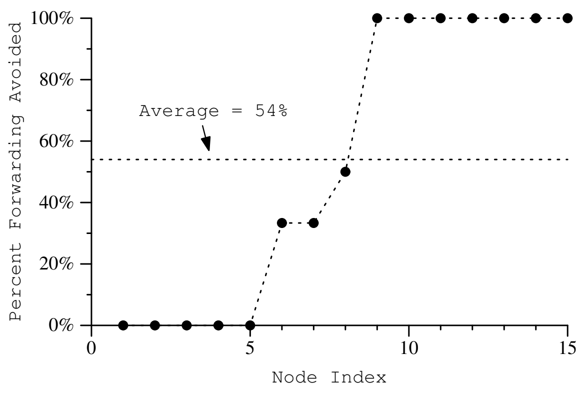

Figure 8: *The fraction of forwarding load avoided if a node adopts a signal strength cheating strategy. (We assume forwarding load is proportional to the number of neighbors.)*

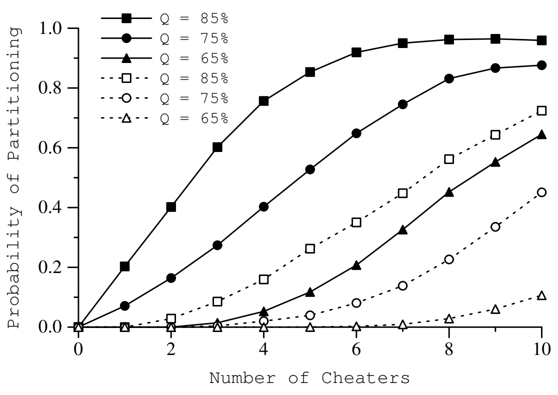

Figure 9: *Probability that the cooperative nodes are partitioned versus varying numbers of (randomly chosen) cheating nodes when running with (dotted lines) and without (solid lines)* Catch. *Under* Catch *the cheaters use a signal strength based attack. Only links with delivery rates at least Q are considered useful.*

or twice. In fact, before moving to a PC-based testbed for OS reliability reasons, we had successfully experimented with Catch on a testbed composed of 10 iPAQs.

5.2.5 Attacks on Anonymity

Catch relies on the ability of nodes to send anonymous messages. If a testee can undermine this property, Catch may be compromised. At the MAC level, anonymity is a reasonable assumption, since it is possible to send packets with an arbitrary source address and contents using commonly available 802.11 hardware [38, 8]. At the physical level, however, strong anonymity cannot be guaranteed against a determined adversary: the source of a packet might be estimated, or at least classified, from the wireless signal's strength or direction. Empirical reports of wireless network conditions [37, 39, 22] and localization schemes based on received signal strength [6] illustrate the difficulties that are involved, however.

Signal strength cheats are a level of escalation beyond the attacks we have defended against thus far.[4] Nonetheless, in this section we study the potential leverage of such attacks. We show that even in its present form Catch is useful in protecting the cooperative nodes and is by far preferable to doing nothing. Taking specific steps in Catch to discourage signal strength based attacks is the subject of future work.

To better understand the threat posed by this kind of attack we consider a cheater that uses signal strength to differentiate among its neighbors, with the goal of connecting to as few of them as possible. The cheater does this by listening to all packets on the air for a short period of time, measuring their signal strengths and sources (for data packets). It then chooses a signal strength threshold for incoming packets that allows it to appear cooperative to neighbors whose packets arrive with strengths above the threshold, and to appear to be a legitimate non-neighbor to all other nodes. It does this by simply ignoring packets with strengths below the threshold and acting cooperatively to those above.[5] Using this procedure, a cheater may end up cooperating with between just one and all of its legitimate neighbors. The cheater prefers the former, but, when Catch is run, overlap in the received signal strengths from multiple neighbors may force it to admit its presence to all of them.

Figure 8 shows the benefits that would be gained by cheating in our testbed using this signal strength approach. For each of the 15 nodes we plot the fraction of forwarding traffic that would be avoided, assuming that forwarding loads are proportional to the number of neighbors, or zero if a cheater manages to establish only a single neighbor. Just under half the time a cheater can escape forwarding entirely, while just over half it avoids none or only a modest amount. Of course, if no protocol is run to protect against cheating, all nodes can cheat 100%, leading to a tragedy of the commons.

Even though a cheater may expect to reduce its forwarding load by about half using signal strength information, running Catch still provides benefits to the cooperative nodes. Figure 9 shows that Catch greatly improves con-

[4]Measuring signal strength required modifications to the network interface driver that enable it to output signal strength data simultaneously with receiving and sending application traffic. The driver-specific modifications to gain access to the signal strength of individual packets raises the barrier to cheating, which is no longer a simple matter of installing a firewall rule. Our hardware cannot give information about signal source direction, nor can any commodity hardware (fitted with an omnidirectional antennae) of which we are aware.

[5]In theory, the cheater can pick an arbitrary signal strength range rather than limiting itself to the top end. But our measurements show that the degree of overlap among neighbors in the middle and bottom part of the range would preclude this behavior. Additionally, better signal strength roughly translates to better connectivity, providing an incentive to pick such neighbors.

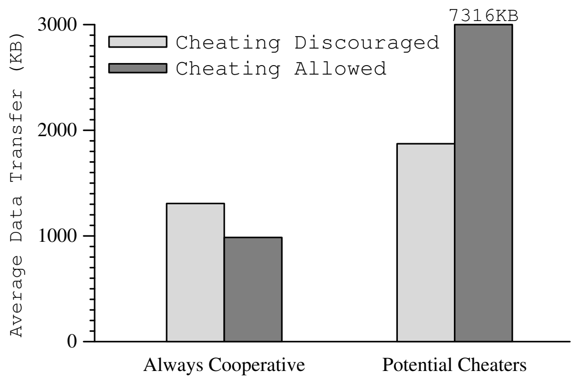

Figure 10: *Average amount of data transferred per node when cheating is discouraged and when it is not. For the former, we assume that no nodes cheat. For the latter, nodes 7, 14 and 15 make use of the communal resources but do not forward any packets. The data transferred for the cheaters is 7316KB with the latter policy.*

nectivity for those nodes, relative to taking no measures against cheating. It plots the probability that a (randomly selected) set of such cheaters would partition the cooperative nodes when running with and without Catch. Because Catch forces many cheaters to admit to multiple neighbors, and so to be available for packet forwarding, it significantly reduces the odds that the network is partitioned. For example, when 20% (3) of the nodes cheat, that probability is lowered from about 60% to about 10% when using the highest quality links. At a 75% link delivery rate threshold, the odds of network a partition are reduced from about 30% to zero.

5.2.6 Summary Effect on Performance

We conclude this section by illustrating the performance benefits of implementing a policy that discourages cheating. Without such a policy, any node that cares to do so can simply fail to forward packets while itself presenting a forwarding load to others. In contrast, if an effective cheating policy is implemented it is self-defeating to cheat, and so we can reasonably expect all nodes that care to connect to the network to cooperate.

In this experiment, we randomly selected 3 nodes as cheaters. All nodes were trying to download randomly selected files from randomly selected servers. Figure 10 illustrates the average amount of data transferred under the two scenarios – “cheating discouraged,” which results in all nodes behaving cooperatively, and “cheating allowed,” where nodes inclined to cheat do so. Both scenarios were run for 35 minutes. The two bars on the left average the per-node results for twelve nodes that acted cooperatively in both scenarios, while the bars on the right average results for the three cheaters (nodes 7, 14, and 15). The right bar for each group gives the average data transferred in a system that does not protect against cheating. The left bar gives the data transferred when policy induces all nodes to behave cooperatively.

This data illustrates two important points. First, there is a very large incentive to cheat: the cheaters improve their throughput by 400% relative to when they are forced to cooperate. This indicates that there is considerable potential motivation for nodes to behave selfishly in these environments if they can do so without retribution. Second, the improved situation for the cheaters comes at the expense of cooperative nodes. The performance of the cooperative nodes is decreased by 25% when 20% of their fellow nodes cheat.

6 Extended Analysis

In this section we extend our analysis of Catch, using simulation to explore issues difficult or impossible to address in the testbed.

6.1 Simulation Testbed and Metrics

We built a simulator to perform this analysis. This simulator does not model the details of packet delivery, but instead generates packet loss and reception counts for each epoch and uses them to drive the protocol state machine. The protocol state machine is parameterized by the neighborhood topology, its loss rates, the z and sign statistical test parameters, and the structure of the isolation FSA. We focus on packets that are subject to Catch’s statistical tests and ignore other (control) packets. Our basic setting includes a single cheater with 6 neighbors. The epoch duration in the simulations is one minute. We set the confidence levels for the z and sign tests (Section 4.2) to 99.999% and 99.995% respectively.

To assess the effectiveness of Catch, we use *Average Time to Isolation* (ATI) as the metric. ATI is measured in units of epochs. An ideal policy would exhibit ATI values of 1 for nodes that cheat (at any rate), and infinite ATI values for those that do not.

6.2 Physical Environment Effects

We first evaluate Catch’s robustness to two characteristics of the physical environment, packet loss and network density. To model cheating, use a straightforward strategy in which the cheater drops data packets randomly with fixed probability. Because the packet losses due to the wireless network are also modeled as a random process, this cheating strategy is arguably difficult for our statistical tests to detect.

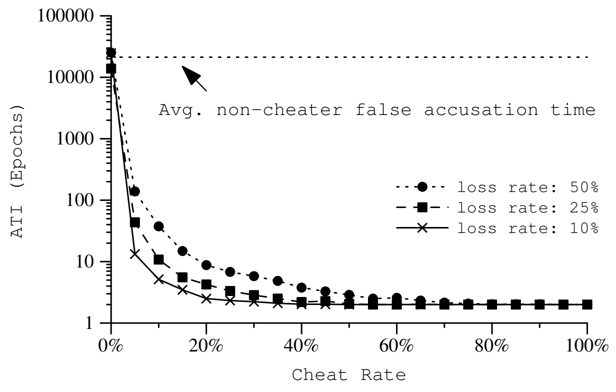

Figure 11: *Average time to isolation versus cheat rate, for various background network loss rates.* (Y-axis on a log scale.)

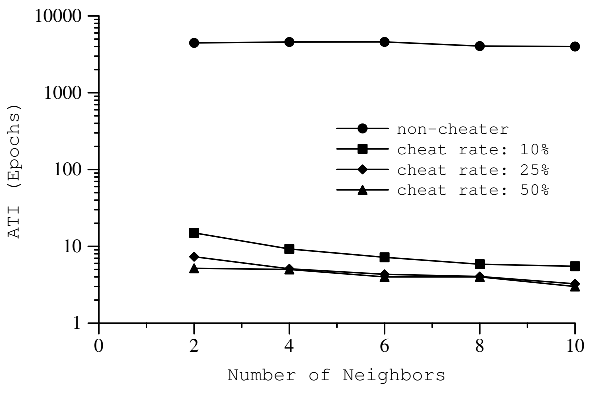

Figure 12: *Average time to isolation versus number of neighbors.* (Y-axis on a log scale.)

6.2.1 Packet Loss

We would expect higher wireless loss rates to make it more difficult to detect cheating. Figure 11 shows ATI results as a function of cheat rate for three different background network loss rates. Each data point shows the average of 40 runs. When there is no cheating (the y-axis), there is a large isolation time – an average of around 26,000 epochs (about 18 days). These times fall steeply as the cheat rate grows, to under 10 epochs for cheat rates of 10-20%. The results for loss rates in the range of 10%-25% are in line with those observed in our testbed (Figure 6), except that the homogeneous link qualities in the simulation environment result in much longer false accusation times than in the more variable testbed. Thus, the impact of high wireless loss rates on Catch is quite small. Even at a loss rate of 50% Catch isolates a cheater that drops 25% of the packets it needs to forward in 7 epochs on average, which is only 4 epochs more than the smallest possible.

6.2.2 Network Density

We would expect Catch to perform better in higher density networks because larger neighborhoods are more likely to make correct statistical decisions. Figure 12 examines the impact of the number of neighbors on detection and false accusation times. We show results for non-cheaters (the top line) as well as for cheating at rates from 10-50%. Increasing the number of neighbors from 6 to 10 yields a small decrease in the time to detect cheaters, as might be expected: already at 6 neighbors there is little room for improvement. More surprisingly, reducing the number of neighbors by a factor of three, to only two, increases detection time by only a few epochs. Additionally, the rate at which non-cheaters are falsely accused is essentially unaffected over the entire range. Thus, Catch seems to be robust, working well in both high and low density networks.

6.3 More Sophisticated Cheaters

We now consider the issue of robustness to more complex attacks. Thus far, we have analyzed a simple cheating model in which the cheater randomly drops data packets it is meant to forward. Here, we use our knowledge of the statistical tests to construct cheating variations that target potential weaknesses. While we cannot prove the negative result that there are no attacks that might be effective against Catch, we can show that these attacks yield only very limited success.

One variation is targeted cheating, in which the cheater drops packets from a time-varying subset of neighbors, rather than uniformly from all. This stresses the z test in Catch, whereas we know that the basic cheater is most often detected by the sign test. We call this approach “rotation.” A second variation attacks the FSA used to make isolation decisions. Since three consecutive failed epoch tests are required to isolate a node, a cheater may attempt to escape isolation by cheating on, say, alternate epochs, keeping the FSA out of its decision state. We call this the “on-off” strategy. Finally, both attacks may be used at once.

Figure 13 plots the number of epochs to isolation for these strategies against the number of nodes targeted, for the difficult environment where the loss rate is as large as the cheat rate. (Both were set to 20%.) The graph suggests that these custom-built strategies are only very modestly successful. The most effective strategy for the cheater is to obtain its overall average cheat rate of 20% by dropping 60% of the packets from 2 of its 6 neighbors, while rotating that pair each epoch. Using that strategy, the cheater is isolated in 9 epochs on average, compared to 5 epochs for the base cheating strategy.

As another variation of the basic cheating model, we experimented with cheaters that drop packets in a deterministic pattern, rather than randomly. The threat here is that

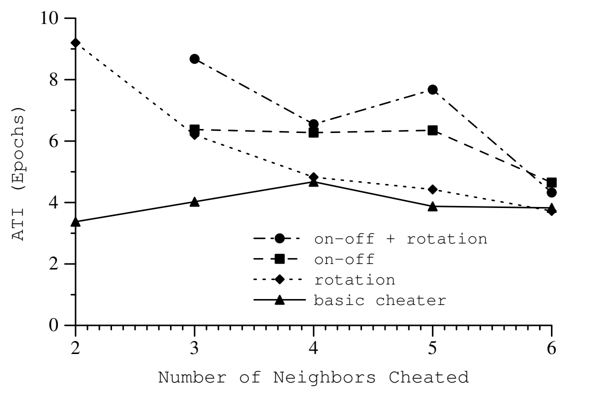

Figure 13: *The time to isolation for customized cheater strategies. The cheater directs all misbehavior in a single epoch to the number of neighbors given on the x-axis.* (Total cheat rate = 20%. Network loss rate = 20%.)

the reduction in variance will help cheaters avoid detection. In fact, the opposite happened: Catch was more effective.

6.4 Assessing Effectiveness

To complete this section, we consider how much better it might be possible to make Catch. This is a difficult question to answer. We address it by comparing Catch to an unrealistically powerful alternative, the *Detection Oracle*, that serves as an informal upper bound on what might be possible by any technique.

The Detection Oracle hears all packet transmissions everywhere in the network, without loss, and so has reliable knowledge of all externally visible events. Additionally, it retains infinite history information, enabling it to apply the Catch statistical tests over this maximal pool of data.[6] In contrast, the nodes in any real system have only imprecise information (due to losses), each one is directly aware of only a subset of the global information, and history information must be devalued because the highly dynamic environment is likely to produce anomalous periods that must eventually be forgiven. For these reasons, we believe the Detection Oracle provides a useful heuristic upper bound on what a real system could achieve.

Figure 14 compares the Detection Oracle with Catch. This graph suggest that Catch does nearly as well as possible. The oracle's advantage exceeds a 5 epoch reduction in detection time only in the case of high network loss rate (50%) and relatively low (5-25%) cheat rates.

[6]The oracle is also provided with knowledge that average loss and cheat rates in the model do not vary over time, something not known to Catch.

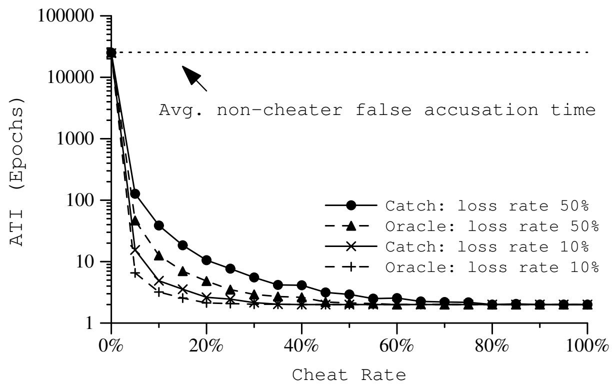

Figure 14: *Comparison of the time to isolation with* Catch *and the Detection Oracle as a function of cheat rate for 10% and 50% network loss rates.* (Y-axis on a log scale.)

7 Related Work

Approaches to discourage selfish behavior in wireless networks can be classified into three broad categories.

Incentive-based approaches discourage cheating by making compliance more attractive. Nodes accumulate virtual currency or credits by forwarding for others, which they can then use for sending their own packets. Examples include Nuglets [12], Sprite [40] and priority forwarding [32]. These schemes rely on a trusted central authority[7] to ensure the integrity of the currency, and to redistribute wealth in the network so that even nodes that are not in a position to forward for others can send their packets. Catch does not rely on any central authority and its operation is completely distributed. Incentives also fail to encourage forwarding in nodes that have very little data of their own to send. This can lead to a disconnected network when light-senders are located at strategic points in the topology.

Game-theoretic approaches formulate the forwarding decision such that forwarding at a certain rate becomes the Nash equilibrium [19] for the network. This means that deviation from the recommended forwarding behavior can only result in situations that are worse for the deviant node. Generous Tit-for-Tat (GTFT) is an example of such an approach [35]. Like GTFT, Catch relies on the mechanics of Tit-for-Tat by assuming cooperation and punishing cheaters. However, while GTFT requires knowledge about the utilities of all the nodes in the network, Catch relies only on information collected in the one-hop neighborhood of individual nodes.

Catch belongs to the class of enforcement-based mechanisms that discourage cheating through the fear of detection and punishment. Part of our detection mechanism – snooping wireless transmission of neighbors –

[7]For schemes employing per-node trusted hardware, we consider the central authority to be the maker of that hardware.

was originally proposed by Marti *et al.* [27], and also used by CONFIDANT [11]. It is our use of it in conjunction with anonymity to detect misbehavior that is novel. We also note that existing enforcement-based protocols [27, 11, 10] rely on reputation spreading to deal with cheating nodes. This requires global flooding, while Catch limits information spread to single-hop neighborhoods. Moreover, simple flooding requires network redundancy – cheating nodes will not forward reputation packets that implicate them. Catch uses anonymity and one-way hash functions to reliably communicate with the neighbors of cheating nodes.

8 Conclusions

We have presented Catch, a protocol to encourage cooperation in multi-hop wireless networks comprised of autonomous nodes. Catch is much more widely applicable than existing approaches, needing no central authority and placing no restrictions on workloads, routing protocols or node objectives. It uses novel strategies based on anonymous messages and statistical tests to detect cheaters with high likelihood and punish them with periods of isolation. *Anonymous challenge messages* are used to estimate true loss rates, even when dealing with untrusted and uncooperative nodes. *Anonymous neighbor verification* is used to compel a node to forward packets, even when the data being carried is contrary to its interests. These techniques are central to Catch, and we hope they will be useful in other applications as well.

We implemented Catch in Linux and performed what is to our knowledge the first evaluation of cooperative routing protocols in an 802.11 wireless testbed. We showed that Catch works well despite volatile wireless conditions and requires little bandwidth overhead (and negligible CPU overhead). In our experiments, cheaters are quickly isolated from the network (and more rapidly for more egregious cheating) and cooperative nodes are rarely accused of cheating. Simulations confirm this finding over a wide range of conditions. We quantified the impact of cheating by showing that the presence of even a few cheaters can partition the network. In one experiment, their presence led to a 25% overall performance degradation for the cooperative nodes. We also explored the leverage of signal strength attacks, and found that even without any measure to actively thwart such attacks, Catch provides worthwhile protection. Extending Catch to defeat these attacks is part of our future work.

References

[1] E. Adar and B. A. Huberman. Free riding on gnutella. *First Monday*, 5(10), 2000.

[2] A. Adya, *et al.* A multi-radio unification protocol for ieee 802.11 wireless networks. Technical Report MSR-TR-2003-41, Microsoft, 2003.

[3] D. Aguayo, *et al.* MIT roofnet implementation. http://www.pdos.lcs.mit.edu/roofnet/design/, 2003.

[4] D. Aguayo, *et al.* A measurement study of a rooftop 802.11b mesh network. In *ACM SIGCOMM*, 2004. To appear.

[5] A. Akella, *et al.* Selfish behavior and stability of the internet: A game-theoretic analysis of tcp. In *ACM SIGCOMM*, 2002.

[6] P. Bahl and V. Padmanabhan. RADAR: an in-building RF-based user location and tracking system. In *IEEE INFOCOM*, 2000.

[7] N. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *IEEE INFOCOM*, 2001.

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

[9] P. Bhagwat, B. Raman, and D. Sanghi. Turning 802.11 inside-out. In *HotNets-II*, 2003.

[10] K. A. Bradley, *et al.* Detecting disruptive routers: A distributed network monitoring approach. In *IEEE symposium on security and privacy*, 1998.

[11] S. Buchegger and J.-Y. L. Boudec. Performance analysis of the CONFIDANT protocol: Cooperation of nodes — fairness in dynamic ad-hoc networks. In *MobiHOC*, 2002.

[12] L. Buttyan and J. Hubaux. Stimulating cooperation in self-organizing mobile ad hoc networks. *ACM/Kluwer Mobile Networks and Applications*, 8(5), 2003.

[13] B.-G. Chun, *et al.* Characterizing selfishly constructed overlay routing networks. In *23rd IEEE International Conference on Computer Communications*, 2004.

[14] D. S. J. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Mobicom*, 2003.

[15] D. S. J. D. Couto, D. Aguayo, B. A. Chambers, and R. Morris. Performance of multihop wireless networks: Shortest path is not enough. In *HotNets-I*, 2002.

[16] R. Flickenger. *Building Wireless Community Networks, 2nd Ed.* OReilly, 2003.

[17] S. Floyd. Congestion control principles. IETF RFC 2914, 2000.

[18] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4), 1999.

[19] D. Fudenberg and J. Tirole. *Game Theory*. MIT PRess, 1991.

[20] P. Gupta and P. R. Kumar. The capacity of wireless networks. *IEEE Transactions on Information Theory*, IT-46, 2000.

[21] F. Kelly, A. Maulloo, and D. Tan. Rate control in communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49:237–252, 1998.

[22] D. Kotz, C. Newport, and C. Elliott. The mistaken axioms of wireless-network research. Tech. Rep. TR2003-467, Dartmouth, 2003.

[23] K. Lai, M. Feldman, I. Stoica, and J. Chuang. Incentives for cooperation in peer-to-peer networks. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[24] S. H. Low, F. Paganini, and J. C. Doyle. Internet congestion control. *IEEE Control Systems Magazine*, pp. 28–43, 2002.

[25] H. Luo, *et al.* UCAN: A unified cellular and ad-hoc network architecture. In *9th Mobicom*, pp. 353–367, 2003.

[26] R. Mahajan, S. Floyd, and D. Wetherall. Controlling high-bandwidth flows at the congested router. In *International Conference on Network Protocols (ICNP)*, 2002.

[27] S. Marti, T. J. Giuli, K. Lai, and M. Baker. Mitigating router misbehavior in mobile ad-hoc networks. In *Mobicom*, 2000.

[28] J. McClave and F. Dietrich. *Statistics*. Macmillan Publishing Company, 6th edn., 1994.

[29] S. Narayanaswamy, V. Kawadia, R. Sreenivas, and P. R. Kumar. Power control in ad-hoc networks: Theory, architecture, algorithm and implementation of the COMPOW protocol. In *European Wireless Conference*, 2002.

[30] T.-W. Ngan, D. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *2nd International Workshop on Peer-to-Peer Systems*, 2003.

[31] M. Pritchard. How to hurt the hackers: The scoop on internet cheating and how you can combat it. http://www.gamasutra.com/features/20000724/pritchard_pfv.htm, 2000.

[32] B. Raghavan and A. C. Snoeren. Priority forwarding in ad hoc networks with self-interested parties. In *Workshop on Peer-to-Peer Systems*, 2003.

[33] S. Saroiu, K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking*, 2002.

[34] S. Shenker. Making greed work in networks: a game-theoretic analysis of switch service disciplines. *IEEE/ACM Transactions on Networking*, 3(6):819–831, 2003.

[35] V. Srinivasan, P. Nuggehalli, C. F. Chiasserini, and R. R. Rao. Cooperation in wireless ad hoc networks. In *IEEE INFOCOM*, 2003.

[36] Q. Sun and H. Garcia-Molina. Slic: A selfish link-based incentive mechanism for unstructured peer-to-peer networks. In *24th International Conference on Distributed Computing Systems*, 2004.

[37] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Sensys*, 2003.

[38] J. Wright. Detecting wireless LAN MAC address spoofing. http://home.jwu.edu/jwright/papers/wlan-mac-spoof.pdf, 2003.

[39] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *SenSys*, 2003.

[40] S. Zhong, Y. Yang, and J. Chen. Sprite: A simple, cheat-proof, credit-based system for mobile ad hoc networks. In *IEEE INFOCOM*, 2003.

A Catch Fail-safes

In this appendix we briefly reconsider each step of Catch in light of possible, intentional violations by the cheater. Our goal is to show that a cheater cannot defeat the protocol by manipulating messages in unanticipated ways. We assume that there is no collusion between cheating nodes.

Epoch-Start. Each node must periodically send EpochStart messages or it is deemed uncooperative by its neighbors and is ignored.

Packet Forwarding and Accounting. The testee can drop some or all of the challenges. However, we argued in Section 3.1 that because the challenges are anonymous it cannot selectively inflate the loss rate on just some links, and has to waste its own resources if it chooses to uniformly inflate the loss rate on all links.

Anonymous Neighbor Verification Open (ANV1). The testee can drop some fraction of the ANV1 messages it receives. However, we argued in Section 3.2 that because of anonymity this will result in the detection in a reasonably short time.

Tester Information Exchange. The testee is unable to interfere with the exchange because it is relying on all of the testers to release their tokens.

Epoch Evaluation and ANV Close (ANV2). It is in the testee's best interest to forward these messages, since they are required for it to pass the epoch evaluation.

Isolation Decision. The tester must drop the cheater's data packets to isolate it. To prevent this punishment from being circumvented we require that some notion of long-lived identity be present in the system and transmitted with the data packets, so that the cheater cannot simply assume another identity.

Deliberate False Accusations A different style of attack is for a cheating *tester* to falsely accuse a cooperative testee of cheating, causing it to be isolated. This could be attractive because the cheating node would no longer need to relay packets for its isolated neighbor. To discourage this, we require cooperative testees that are unfairly isolated to cause isolation of their attacker so that the attacker looses network connectivity. This is the well-known TIT-FOR-TAT strategy that is often used to encourage cooperation [19].