CRAFT: A New Secure Congestion Control Architecture

By

Dongho Kim
Jerry T. Chiang
Yih-Chun Hu
Adrian Perrig
P. R. Kumar

Technical Report
UILU-ENG-2010-2511

Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Urbana, Illinois
September 2010

CRAFT: A New Secure Congestion Control Architecture

Dongho Kim* dkim99@illinois.edu
Jerry T. Chiang* chiang2@illinois.edu
Yih-Chun Hu* yihchun@illinois.edu
Adrian Perrig† perrig@cmu.edu
P. R. Kumar* prkumar@illinois.edu

Technical Report

ABSTRACT

Congestion control algorithms seek to optimally utilize network resources by allocating a certain rate for each user. However, malicious clients can disregard the congestion control algorithms and induce congestion at bottleneck links. Thus, in an adversarial environment, the network must enforce the congestion control algorithm in order to attain the optimal network utilization offered by the algorithm.

Prior work protects only a single link incident on the enforcement router, neglecting damage inflicted upon other downstream links. We present CRAFT, a capability-based scheme to secure *all* downstream links of a deploying router. Our basic approach is to enforce a network-wide congestion control algorithm on all flows. As a reference design, we develop techniques to enforce the TCP congestion control. Our design regulates all flows to share bandwidth resources in a TCP-fair manner by emulating the TCP state machine in a CRAFT router. As a result, once a flow passes a single CRAFT router, it is TCP-fair on all downstream links of that router. Our prototype implementation shows that CRAFT provides strong fairness properties with low overhead.

1. INTRODUCTION

Congestion control algorithms seek to optimally utilize network resources by allocating a certain rate for each user. The optimal utilization is often defined using some fairness metric, including max-min fairness and proportional fairness [15]. Researchers have extensively studied the congestion control problem, including end-to-end approach [1], router-assisted approach [12], adaptive studies to diverse network environment [3, 10], and theoretical studies [15, 19]. While most work in congestion control assumes that all entities follow the rules specified by a congestion control algorithm, some work has also considered an adversarial environment where a misbehaving user does not follow the specified rule [22, 23], thereby gaining network resource allocation that exceeds his fair share or suppressing the network resource allocated to others.

Some work adopts a reactive approach to defend against misbehaving users. Floyd et al. [11] use a TCP throughput equation to determine the proper throughput for a flow and categorize any flow using more than its fair throughput as "not TCP friendly". Stoica et al. [26] estimate the rate of a flow and assign that flow a fair share of a link according to the estimated rate. In contrast to the protocols proposed by Floyd et al. and Stoica et al., fair queueing [7] is a preventive approach where a router allocates a fixed amount of bandwidth to a flow or the aggregate of several flows when that router experiences congestion.

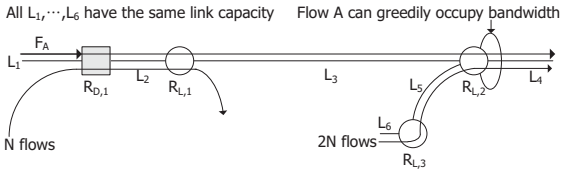


Figure 1: An example that illustrates single-link protection is insufficient for protecting downstream links.

While these studies have desirable properties in some contexts, they are all *single-link* schemes. That is, a router deploying any of the proposed defense mechanisms can protect only the link immediately behind that router. Consequently, a single-link congestion control scheme must be deployed at each network bottleneck. If the network topology is complex and dynamic, a single-link congestion control scheme may require universal deployment in order to protect the entire network from misbehavior.

We now illustrate how a single-link protection mechanism is not sufficient to enforce fair sharing of throughput throughout the entire network. Figure 1 shows a network in which all links have the same link capacity. Let there be a flow that traverses multiple links together with different number of flows at each link. The bandwidth-fair rate of a flow traversing a link is simply the bandwidth of the link divided by the number of flows sharing the link. In this example network, flow F_A and N other flows traverse router $R_{D,1}$ that deploys a single-link protection mechanism, enforcing that each flow gets $\frac{1}{N+1}$ of the link capacity. That is, F_A shares link L_2 with N other flows in a bandwidth-fair manner. Suppose F_A's destination is different from that of the other N flows; then F_A, but not any of the N other flows, would traverse the legacy router $R_{L,2}$. If $2N$ other flows also traverse through the router $R_{L,2}$, F_A could obtain roughly twice as much bandwidth as each of the other $2N$ flows could.

In this paper, we consider *network-wide* protection against a misbehaving flow. That is, a router deploying a defense mechanism can protect all its downstream links (regardless of the number of hops between the router and the link) from a misbehaving flow after the flow traverses the router. A network-wide protection scheme is desirable in networks where full-deployment of a defense mechanism is difficult or costly. Intuitively, a router cannot protect the network from flows the router has not seen; therefore, protecting all its downstream links provides optimal protection achievable by a router.

We present CRAFT (Capability-based Regulation of All Flows and Traffic), a secure congestion control architecture that provides network-wide protection. Our basic approach is to enforce that all

*Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign
†Department of Electrical and Computer Engineering, Carnegie Mellon University

flows follow a specified congestion control algorithm. We let a CRAFT router drop packets of a misbehaving flow so that the flow competes fairly with other flows in the downstream links of the router. Our contributions are:

- We show that our approach is realizable by designing the enforcement of TCP, the most widely deployed congestion control algorithm in the Internet.
- We design CRAFT to provide security while remaining efficient in terms of router and end-host computation and storage.
- We implement CRAFT to show that our design is practical and deployable.

We start our paper with our problem statement.

2. PROBLEM STATEMENT

2.1 Goal and Basic Approach

In this paper, we present a secure protocol that provides network-wide protection. After a flow that claims to comply with our protocol traverses a single deploying router, that flow would fairly share all downstream links with other legitimate and legacy TCP traffic. Our proposed protocol achieves this network-wide protection property by *emulating the state machine of each flow and enforcing that each flow follows a specified congestion control algorithm*. We will consider the overhead of emulating the state of each flow in our design. We believe it is crucial to investigate network-wide protocols so that deployment cost is not prohibitive for the system operators in funding-constrained networks. To make our discussion concrete, we design our protocol to *emulate the TCP state machine* and enforce TCP-fairness.

2.2 Design Consideration

To efficiently and practically emulate the TCP state machine, we list several crucial design considerations in this section. We also briefly cover how our proposed protocol deals with these design considerations.

- **Fairness:** There can be various definitions of fairness. In this paper, we are most concerned with a network-wide protocol, we thus consider a fair rate to be the rate that is obtainable by a legitimate TCP flow since TCP is the predominant congestion control protocol used in the Internet today.
- **Minimal trust:** If a router trusts any entity besides itself to behave correctly, an attacker can compromise the trusted entity and attack the services provided by that router. To provide the strongest security, we assume that a CRAFT router trusts only other routers in the same autonomous system (AS) as itself. Specifically, a CRAFT router does not use information provided by any router outside its AS, the sender, or the receiver to bootstrap security.
- **Asymmetric route:** Generally, the Internet path of a given flow is not symmetric [13]. Hence, we assume that, although a downstream packet goes through a particular CRAFT router, the corresponding upstream packet (acknowledgment) might not go through the same CRAFT router.
- **Minimal per-flow state at routers** Since a router only has limited amount of memory, a router must minimize the amount of per-flow state stored in order to service the maximum number of flows. Our protocol uses pre-capability to lessen the per-flow state stored at a router.
- **Packet reordering or loss:** In the Internet, packets can be reordered or lost. Identifying packet reordering or loss is important for emulating the TCP state of a flow since packet reordering and loss affect a benign TCP receiver's acknowledgment behavior thereby affecting the rate of a benign TCP flow. Our proposed CRAFT header allows a router to identify packet reorderings and losses similar to a TCP client.
- **Other transport layer protocols:** We use TCP fairness as our definition for fair congestion control. Applications use UDP for various purposes, e.g. to avoid retransmissions, to avoid rate control in the event of packet losses, and to reduce the transport-layer overhead of each packet. Nonetheless, in a network that seeks to prevent denial-of-service attacks, a flow must be subjected to congestion control. Therefore, in our design, non-TCP flows are rate-limited to the TCP-fair rate.
- **Various flavors of TCP:** Since the TCP standard leaves much room for interpretation, many TCP variants are used in the Internet today. One way to deal with different flavors of TCP is to have each flow specify the TCP variant used in the packet header and try to emulate that TCP variant. The other way is to emulate the most aggressive TCP that is still acceptable. In this paper, we enforce an upper bound of the TCP congestion window that is compliant to RFC 2581. We discuss the issues involved in supporting specific flavors of TCP in Appendix C.1.

2.3 Threat Model

We design our protocol to treat the above design considerations under a strong attacker model. We make no assumptions about whether the routers or end hosts are compromised; in particular, we do not prevent compromised end hosts from colluding. A compromised end host or router can send packets at any arbitrary rate, eavesdrop, modify, and drop any packets that are generated by or routed through them.

3. STRAWMAN DESIGN

To better understand CRAFT, we present a '*strawman*' design to securely emulate the TCP state of a flow before we present our protocol. We let a flow be defined as a 4-tuple: source IP, source port, destination IP, and destination port. Our strawman design illustrates a method that emulates the TCP congestion control protocol [1] with high overhead in an idealized environment where all packets reach the destination in order, and the path of the flow might be asymmetric but does not change over the lifetime of the flow. We provide careful treatment on how CRAFT relaxes these limitations in Section 4. Detailed explanation of the TCP congestion control [1, 24] is beyond the scope of this paper, and is summarized in Appendix A.

The TCP congestion control assumes that a pair of sender and receiver would behave in a manner specified in the protocol; specifically, the sender would increase the rate of the flow only when the the receiver honestly acknowledges receiving a packet. However, in an adversarial environment, a sender can arbitrarily increase its rate without having received any acknowledgments. Furthermore, a receiver can send acknowledgments without actually having received a packet [22] since acknowledgments in the TCP are not cryptographically dependent on the data in the TCP packet.

Our strawman design illustrates how we can prevent a TCP flow from misbehaving. For simplicity, we present our proposed strawman and CRAFT protocols as each would operate when data flows in only one direction, even though our protocol designs, simulation, and implementation extend readily to bidirectional flows.

Intuition. The main objective of our strawman design is to ensure that a downstream packet is indeed received by a receiver and the

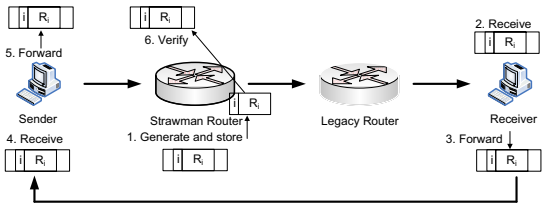


Figure 2: Strawman design: a strawman router generates a Random Value (R_i) corresponding to a Packet ID (i) and stores the Random Value. After the receiver gets the Random Value, the receiver forwards the Random Value to the sender. In a future packet, the sender includes the received Random Value with the data. When the packet reaches the strawman router, the router compares the received Random Value to the stored Random Value, thereby ensuring that the previous packet and its acknowledgment were successfully received by the receiver and the sender, respectively.

Algorithm 1 Strawman version

Packet header from sender: <Packet ID (pid), Random Value ID (rid), Random Value (R_{rid})>
On strawman router's receiving a packet
$r \leftarrow$ stored_random_value(rid)
if $r == R_{rid}$ **then**
  calculate_new_window_size()
  $R_{pid} \leftarrow$ generate_new_random_value(pid)
  stored_random_value(pid) $\leftarrow$ (R_{pid})
  insert_random_value_to_packet(R_{pid})
else
  drop packet
end if
Packet header from router: <Packet ID (pid), Random Value (R_{pid})>
On receiver's sending a packet
insert_random_value_to_packet(R_{pid})
On sender's sending a packet
insert_random_value_to_packet(R_{pid})
insert_new_packet_ID_to_packet($pid + 1$)

corresponding acknowledgment is received by a sender. A strawman router can generate a Random Value for each packet to verify that the receiver has received a downstream packet, that the receiver has sent the Random Value back to the sender in its acknowledgment packet, and that the sender has received the corresponding acknowledgment, as shown in Figure 2.

When the strawman router sees a downstream packet, it inserts a Random Value into the packet, routes the packet to the receiver, and stores the Random Value and its associated flow in its memory. The receiver, upon receiving the packet, sends the Random Value back to the sender along with the TCP acknowledgment. The next time the sender wishes to send a packet to the receiver, the sender includes in the packet the Random Value he received that was embedded in the acknowledgment of the previous packet.

Since we assumed that the path from the sender to the receiver does not change, the data packet including the Random Value will reach the strawman router. The strawman router then verifies that the Random Value included in the packet is the same as that stored in the router. Since the Random Value is generated by the strawman router, a matching pair of Random Values in the packet and in the router's local memory implies that the previous packet was successfully received and acknowledged. In other words, the receiver and the sender collaboratively acknowledge to the router that the Random Value and the packets therein were successfully received.

This architecture addresses the *minimal trust* consideration in Section 2.2 since a strawman router is required to trust only itself. Our strawman design also addresses the *asymmetry of Internet path* consideration since our strawman router does not require the upstream packets or the acknowledgment packets to traverse the same strawman router that the downstream packet traversed. Our CRAFT protocol preserves these properties.

Detailed description. Algorithm 1 shows the packet header structure and how the sender, receiver, and a strawman router behave in our strawman design. When a sender wants to send a new packet over a particular flow, it assigns a unique Packet ID to each packet. The sender includes the following data with the packet header: the Packet ID, the latest received Random Value, and the corresponding Random Value ID. The Random Value ID is the Packet ID of a packet for which a strawman router previously generated a Random Value.

When a packet arrives at a strawman router, the router verifies that the included Random Value is valid. That is, the router compares the included Random Value with that stored locally at the router. If the value included in the packet matches that stored locally, the router calculates the new congestion window (cwnd) size for the corresponding flow. We explain this function (calculate_new_window_size()) in more detail in Section 4.4. The router then generates a new Random Value and replaces the Random Value included in the packet. The strawman router then locally stores this new Random Value and Packet ID and routes the packet to its destination. If the value included in the packet is different from that stored locally, the router punishes the corresponding flow by dropping the packet or deleting the entire flow state.

The receiver, upon receiving a packet, forwards the received Packet ID and the Random Value to the sender. The Packet ID and the Random Value can simply piggyback on the acknowledgment.

When the sender receives the acknowledgment and the Random Value associated with a particular Packet ID, the sender can use the Packet ID, the associated Random Value, and the congestion control protocol to send more packets.

Our strawman design allows a strawman router to securely verify that a receiver only acknowledges packets that are successfully received, and a sender only sends more packets if the sender has received acknowledgments of previous packets.

We now discuss two undesirable properties of our strawman design. A strawman router needs to store a Random Value for each outstanding packet so that when an outstanding packet is acknowledged, the router can verify the included Random Value and update the emulated congestion window. Consequently, as the TCP congestion window size of each flow grows over time, a strawman router may need to store a large amount of Random Values.

Our strawman design also does not gracefully handle delayed acknowledgments. In general, a receiver generates one delayed acknowledgment for every other received packets. However, since our strawman packet header only includes one Random Value, a strawman router cannot distinguish between a flow that has no packet losses from a flow that loses every other data packet. A simple modification to handle delayed ack is to include multiple Random Values in the packet header. However, this method increases the size of the packet header.

Both of the mentioned undesirable properties are results of using independent Random Values for each outstanding packet. If a later Random Value depends on a previous Random Value, then a router only needs to verify the latest Random Value to implicitly verify all previous Random Values. We thus introduce the concepts of pre-capability and capability in the next section.

4. CRAFT DESIGN

In this section, we present how CRAFT relaxes the limitations of our strawman design to efficiently emulate the TCP state of each flow in the real Internet environment. Our specific target version of TCP is based on the standard documentation [1], which is summarized in Appendix A. There are many variants of the standard TCP;

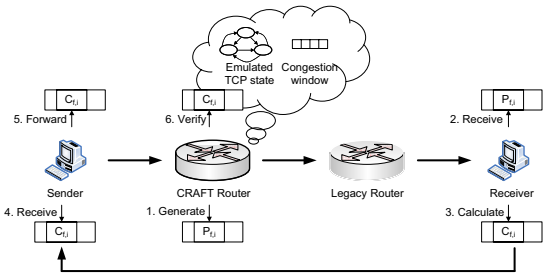


Figure 3: Capability-based enforcement of TCP congestion control algorithm. A CRAFT router generates a pre-capability ($P_{f,i}$) for a packet (i) of a flow (f). After the receiver gets the pre-capability, the receiver calculates and forwards a new capability ($C_{f,i}$) to the sender. The sender includes received capability to the CRAFT router in a future packet.

however, we do not focus on compliance of any particular variant, but instead, on providing techniques to efficiently handle reordering and loss. Since we design CRAFT without any specific TCP variants in mind, CRAFT emulates the congestion window (cwnd) of a flow using the maximum allowed cwnd based on the standard. Usually, TCP variants differ only in the management of cwnd in response to packet losses. Thus, by properly modifying our proposed techniques in emulating the cwnd, CRAFT can be readily adapted for specific subsets of TCP variants.

4.1 Overview

CRAFT operates similarly to our strawman design, we illustrate CRAFT in Figure 3. The main difference between CRAFT and our strawman design is that CRAFT uses *pre-capability* and *capability* instead of a Random Value to verify that a packet and its acknowledgment are received by the receiver and the sender respectively. We design pre-capability and capability so they can gracefully handle reordering and loss in a memory space-efficient manner. We explain the detailed construction of pre-capability and capability in Section 4.2. Algorithm 2 shows the packet header of CRAFT packets and the whole procedure of CRAFT after a connection is established. We explain each field in the packet header when we explain its associated operations.

When a sender initiates a CRAFT flow, it sends a CRAFT request packet, and the router allocates some of its memory to store the state of the new flow. When the flow initiation packet reaches the receiver, the receiver allocates some memory for the new flow, and sends an acknowledgment to the sender to finish the connection establishment. Establishing a connection can be done by using the TCP SYN and SYN-ACK packets.

Once a sender establishes a connection, the sender assigns each of his outgoing packets with a unique and contiguous integer, called the *Packet ID*. The CRAFT Packet ID is different from a TCP sequence number since the TCP sequence number is issued based on the byte number, while the CRAFT Packet ID is issued based on the packet number.

For each packet routed through a CRAFT router, the router verifies that the TCP cwnd of the corresponding flow allows the flow to send that packet. The CRAFT router uses the *ACK ID* and the *capability* in the packet header to update a flow's cwnd. If a packet is sent exceeding the flow's cwnd, the packet is dropped. Otherwise, the CRAFT router generates a *pre-capability* for the packet, inserts the pre-capability in place of the capability in the packet header, and then route the packet to its destination. We explain how a CRAFT router can verify capabilities and generate pre-capabilities in Section 4.2 and Section 4.3. We then explain how a CRAFT router updates the emulated cwnd of a flow in Section 4.4.

Algorithm 2 CRAFT

Packet header from sender: <Packet ID (pid), ACK ID ($ackid$), Capability (C), Timeout (TO), Reordered Packet IDs ($rids$), Lost Packet IDs ($lids$)>
On CRAFT router's receiving a packet
if check_duplicate_packetID(pid) **then**
  drop packet
end if
$c \leftarrow$ calculate_capability($ackid, rids, lids$) (Section 4.2 and Section 4.3)
if $c == C$ **then**
  calculate_new_window_size() (Section 4.4)
  if is_packet_acceptable() **then**
    $P \leftarrow$ calculate_pre-capability(pid) (Section 4.2)
    insert_pre-capability_to_packet(P)
  else
    drop packet
  end if
else
  drop packet
end if
Packet header from router: <Packet ID (pid), ACK ID ($ackid$), Pre-capability (P), Reordered Packet IDs ($rids$), Lost Packet IDs ($lids$)>
On receiver's receiving a packet
$C \leftarrow$ calculate_capability() (Section 4.2 and Section 4.3)
store_capability(C)
Packet header from receiver: <Packet ID (pid), ACK ID ($ackid$), Capability (C), Reordered Packet IDs ($rids$), Lost Packet IDs ($lids$)>
On receiver's sending a packet
insert_capability_to_packet(C)
On sender's receiving a packet
store_capability(C)
On sender's sending a packet
insert_capability_to_packet(C)
insert_new_packet_ID_to_packet($pid + 1$)

To reduce the network traffic, a CRAFT router can use the Packet ID of a packet to remove duplicate packets similar to how IPSec detects duplicates. For each flow, a CRAFT router keeps a list of the Packet IDs of a number of the flow's most recent packets. If the Packet ID of a packet indicates that the packet is a duplicate, the CRAFT router discards the packet.

4.2 Pre-capability and Capability

In this section, we present how a CRAFT router generates the pre-capability associated with a packet and verifies the capability in the packet header. Specifically, we show how pre-capability and capability reduce the memory space overhead while preserving the security features of our strawman design.

In order to clearly present CRAFT, we do not consider any packet reordering or loss in this section. In case a packet is reordered or lost, our construction of pre-capability and capability remains unchanged, only the router's capability verification mechanism is altered. We explain how to handle reordering and loss in Section 4.3.

Intuition. The inefficient use of memory space and the difficulty of handling delayed ack in our strawman design result from the fact that a Random Value inserted into a packet is unrelated to that packet's Packet ID and any previously inserted Random Values. We thus let a CRAFT router insert into each packet a secret *pre-capability*, which is calculated using the Packet ID. We then define the *capability* of a flow to be an aggregate of all the past pre-capabilities inserted into its packets. Both the CRAFT router and the receiver of the flow can calculate the capability since the CRAFT router can generate all the pre-capabilities, and the receiver can store all the seen pre-capabilities.

Since the pre-capability of a packet is calculated using the Packet ID, a CRAFT router should not store all previous pre-capabilities issued to a flow in order to conserve the limited memory space of a router. However, without the router storing the pre-capabilities, we must not let the sender and receiver of a flow be able to generate the pre-capabilities before a packet is sent. Otherwise the sender and the receiver can collaboratively defeat the verification mechanism

shown in our strawman design. CRAFT provides unpredictability by using computationally efficient keyed MAC (Message Authentication Code).

As shown in Figure 3, a CRAFT router generates a pre-capability ($P_{f,i}$) when a new packet comes in. A pre-capability is generated by using a cryptographically secure hash function g:

$$P_{f,i} = g(K, f, i),$$

where K is the secret key of the CRAFT router, f is the flow ID, and i is the Packet ID. Flow ID is a large and unique value randomly generated by the router when the flow is created. If g is a hash function in the Random Oracle Model [4], then $P_{f,i}$ is indistinguishable from a random number because the pair (f, i) has not been previously seen. Since K is known to only the CRAFT router, it is computationally inefficient to guess $P_{f,i}$.

We further elaborate on the g function with a telescoping construction:

$$g(K, f, i) = E_K(f||i) \oplus E_K(f||i+1)$$

where E_K represents a computationally efficient keyed MAC such as HMAC [16] and $||$ is the concatenation operator. By choosing E_K as a secure pseudo-random function, it follows that $P_{f,i}$ is also indistinguishable from a random number. The telescoping construction enables the CRAFT router to calculate a capability corresponding to a set of contiguous pre-capabilities in constant time.

When the receiver receives a packet (i) with a pre-capability ($P_{f,i}$), the receiver constructs the capability ($C_{f,i}$) associated with the packet. A capability ($C_{f,i}$) is defined to be the exclusive-or of all received pre-capabilities up to i. If there is no reordering or loss,

$$C_{f,i} = P_{f,0} \oplus \ldots \oplus P_{f,i},$$

By using the exclusive-or function, we can take advantage of a desirable property: if any input and its distribution are unknown, the output is uniformly distributed over the domain, yielding the largest uncertainty and secrecy in the information theoretic sense. Moreover, since $P_{f,i} = E_K(f||i) \oplus E_K(f||i+1)$, the capability can be calculated efficiently using

$$C_{f,i} = E_K(f||0) \oplus E_K(f||i+1).$$

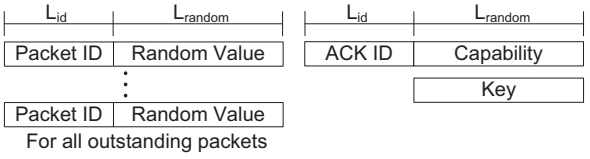
Hence, we need only a single xor operation to calculate the capability associated with a set of contiguous pre-capabilities.

When the receiver sends an acknowledgment to the sender, the receiver inserts the received Packet ID i and the corresponding capability $C_{f,i}$ in the acknowledgment. We call this Packet ID the *ACK ID*. ACK ID is defined to be the Packet ID of the latest *contiguous* packet that a receiver is acknowledging, subject to any previous disclosed losses. A received Packet ID is contiguous as long as there is no packet reordering or loss. We provide careful treatment when considering packet reorders and losses in Section 4.3, and the term 'disclosed loss' will become clear.

When the sender sends a new packet, the new Packet ID will be $i + 1$. The sender then inserts ACK ID i and capability $C_{f,i}$ in the new packet. When the CRAFT router gets the new packet from the sender, it can calculate capability $C_{f,i}$ corresponding to ACK ID i and flow ID f:

$$C_{f,i} = E_K(f||0) \oplus E_K(f||i+1).$$

This operation requires only two hash operations and one xor operation. If a CRAFT router stores $C_{f,i-1}$ and ACK ID i as shown in Figure 4(b), the router only needs to perform one hash operation and one xor operation at the expense of using memory space to store data. The ACK ID and capability are necessary for handling reordering or loss.



(a) Strawman design (b) CRAFT

Figure 4: Comparison of router memory usage between strawman design and CRAFT for a flow: L_{id} and L_{random} are reference length for comparison

We compare the memory space used by our strawman design and the CRAFT protocol in Figure 4. We exclude variables related to emulating TCP state since both designs require the same amount of memory space for these variables. For each flow our strawman router keeps the Packet IDs and the associated Random Values for all outstanding packets. On the other hand, for each flow, a CRAFT router keeps only a secret key, the ACK ID, and the capability regardless of the number of outstanding packets. This property results from the fact that a CRAFT router can calculate the capability with the Packet ID included in the packet header. For handling delayed ack, it is important to note that $C_{f,i}$ proves that the receiver has successfully received not only packet i but also all the packets up to i. Hence, the receiver can acknowledge two packets at once (Packet ID i and $i+1$), and the sender can subsequently prove so to the router, by using only one capability $C_{f,i+1}$. This accumulative property of capability enables CRAFT to use the minimal number of capabilities to handle delayed acks.

4.3 Handling Reordering and Loss

Intuition. As long as all packets are successfully received in order, a CRAFT router can easily verify a capability by calculating $C_{f,i} = E_K(f||0) \oplus E_K(f||i+1)$. However, if packets are lost, the receiver cannot learn the pre-capability of the lost packets, and the receiver thus cannot construct the capability. In this scenario, the receiver computes the capability by excluding the pre-capabilities associated with the lost packets, the receiver also discloses in the acknowledgment packet header the packet IDs of the lost packets. The CRAFT router then excludes the pre-capabilities of the lost packets when calculating and verifying the received capability. CRAFT follows the standard TCP where a packet that is out-of-order by too much is considered lost.

Detailed description. CRAFT uses the header fields of Reordered Packet IDs and Lost Packet IDs (shown in Algorithm 2) to handle packet reordering and loss. Reordered Packet IDs are a list of non-contiguous packet IDs that a receiver receives out of order. Lost Packet IDs are a list of Packet IDs that a receiver has not received. For example, consider the case that a sender has received acknowledgments for packets 0 through i, and packet $i + 1$ is lost. When a receiver receives packet $i+2$, the receiver considers packet $i + 2$ as a reordered packet and sends an immediate (not delayed) acknowledgment which contains in its packet header ACK ID of i and Reordered Packet IDs of $i + 2$. This is similar to a partial acknowledgment, which is a modified duplicate acknowledgment issued by the TCP congestion control. The acknowledgment for packet $i + 3$ then contains ACK ID of i and Reordered Packet IDs of $\{i + 2, i + 3\}$. When packet $i + 4$ arrives at the receiver, the receiver issues a third duplicate acknowledgment and concludes that packet ID $i + 1$ is lost [1]. This third duplicate acknowledgment contains ACK ID of $i + 4$ and Lost Packet ID of $i + 1$.

When a receiver sends an acknowledgment, the receiver inserts a capability as defined in Section 4.2. For the acknowledgment of ACK ID i and Reordered Packet IDs $\{i+2, i+3\}$, the capability is

$$C = C_{f,i} \oplus P_{f,i+2} \oplus P_{f,i+3}.$$

For the acknowledgment of ACK ID $i+4$ and Lost Packet ID $i+1$, the capability is

$$C = C_{f,i} \oplus P_{f,i+2} \oplus P_{f,i+3} \oplus P_{f,i+4}.$$

After the sender receives the acknowledgments containing the fields of Reordered Packet IDs or Lost Packet IDs, the sender's new packet simply carries these fields with the received capability to the CRAFT router. One might think that the number of these fields becomes larger over time as more packets are lost. However, since TCP concludes that a packet is lost if three duplicate acknowledgments are issued, at most two Reordered Packet IDs need to be included in a CRAFT packet header. Moreover, since the CRAFT stores the ACK ID and the corresponding capability of a flow, when a receiver receives a packet, the receiver no longer needs to include any Lost Packet IDs that are smaller than the ACK ID in the packet. In the previous example where packet $i+1$ is lost, when the receiver receives a packet with ACK ID $i+4$, the receiver does not need to include $i+1$ in the field of Lost Packet IDs. Additionally, if a block of contiguous packets are lost, a receiver can specify the interval of Lost Packet IDs rather than individual Lost Packet IDs to reduce the length of the packet header.

We present how a CRAFT router verifies the capability of a packet when previous packets are reordered or lost. When the Reordered Packet IDs field in the header of a packet is not empty, a CRAFT router simply verifies that the included capability matches the capability up to the ACK ID included in the packet header xor the pre-capabilities of the reordered packets.

If the Reordered Packet IDs and Lost Packet IDs fields in the header of a packet are not empty, a CRAFT router calculates the capability of a packet in the following manner. If the router has stored ACK ID i and the corresponding capability C, and the new packet indicates ACK ID $j > i$, Lost Packet IDs $\mathbb{L} = \{\ell_1, \ell_2, \ldots, \ell_n\}$, Reordered Packet IDs r_1 and r_2 and capability C', then C' is valid if and only if

$$C' = C \oplus P_{f,r_1} \oplus P_{f,r_2} \bigoplus_{\{x \in ([i+1,j] \setminus \mathbb{L})\}} P_{f,x};$$

that is, when we xor C with the pre-capabilities of the reordered packets and with the pre-capabilities between $i+1$ and j that are not associated with a lost packet, the result should be the new capability[1]. If C' is valid, we update the (ACK ID, capability) pair stored at the router to (j, C').

In our previous example, when packet with ACK ID $i+4$ and Lost Packet ID $i+1$ reaches the CRAFT router, the router verifies that the included capability equals $C_{f,i} \oplus P_{f,i+2} \oplus P_{f,i+3} \oplus P_{f,i+4}$. If so, the new capability is consistent with the last valid capability xor the pre-capabilities between the last valid capability and the current capability, and the CRAFT router updates the stored ACK ID and the capability.

[1]Since $x \oplus x = 0$, we can efficiently calculate

$$\bigoplus_{x \in ([i+1,j] \setminus \mathbb{L})} P_{f,x} = E_K(f||i+1) \oplus E_K(f||j+1) \bigoplus_{x \in ([i+1,j] \cap \mathbb{L})} P_{f,x}$$

4.4 Congestion Window Calculation

When a CRAFT router verifies a new capability, it knows that one or more additional packets have been received and acknowledged by the receiver, so the router updates its emulated cwnd accordingly.

Like a TCP sender, a CRAFT router stores the cwnd, the slow start threshold (ssthresh), and the maximum segment size of each flow in its memory. The sender discloses the maximum segment size (mss) at connection initialization, and whenever the mss changes. The CRAFT router updates this variable in a manner consistent with the behavior of a legitimate TCP sender-receiver pair. The cwnd and ssthresh are initialized to some predefined values suggested by the TCP congestion control.

The CRAFT router can distinguish between acknowledgments generated from contiguous packets and those generated due to out-of-order delivery by examining the ACK ID and the Reordered Packet IDs. If a sequence of packets are acknowledged in order, the CRAFT router emulates a sender that has received one acknowledgment for every packet he sends. This represents the maximum number of acknowledgments the TCP standard allows the receiver to send. Though a unidirectional TCP flow might use delayed acknowledgments, i.e. sending only one acknowledgment for every other received packets, a bidirectional TCP flow opportunistically carries acknowledgments on reverse traffic, allowing the possibility of sending one acknowledgment per packet. Since the cwnd grows faster when the sender sends more acknowledgment, we choose to emulate the cwnd by assuming one-ACK-per-packet in order to form a reasonable upper bound of the cwnd.

When a CRAFT router observes any Reordered Packet IDs, the router concludes that at least one packet is out-of-order. The router does not adjust the emulated cwnd until the router sees advancement in the ACK ID with either none or a number of Lost Packet IDs. When the CRAFT router sees a Lost Packet ID, the emulated state enters fast recovery phase. The CRAFT router reduces the ssthresh to the maximum of half of the amount of outstanding packets or two times the mss and sets the emulated cwnd to the new ssthresh plus 3 mss to take into consideration the three duplicate acknowledgments. The CRAFT router considers the first packet with disclosed packet loss as a retransmitted packet and records that packet's Packet ID. When the CRAFT router sees an ACK ID with increment of 1 during fast recovery phase, it increases cwnd by 1 mss to keep the same number of outstanding packets. When the CRAFT router sees a packet with an ACK ID equal to or larger than the recorded ID of the retransmission packet, the emulated state exits from fast recovery phase and the CRAFT router sets the emulated cwnd to ssthresh.

Under some circumstances, a TCP sender might lose a packet and not recover through the fast retransmission mechanism. In these cases, after a certain period of time, the sender should time out, and set the timeout (TO) field of the next packet header to 1 to reflect recovering from a timeout in CRAFT. A CRAFT router that sees a packet with the timeout bit of its packet header set to 1 then accepts the ACK ID and any capability borne with this packet (to reduce the burden of listing all Lost Packet IDs), resets the cwnd to one packet, and halves the ssthresh in accordance with the TCP congestion control.

4.5 Remaining Details

When CRAFT is deployed over multi-AS (Autonomous System) environment, a CRAFT router in an AS might not trust another CRAFT router in another AS. CRAFT thus uses a separate set of pre-capabilities and capabilities in the packet header for each CRAFT router. To accommodate multiple capabilities, CRAFT can

use short capability that is each a few bits in length. We discuss the security of using short capability in Section 5.1.

When a network wants to deploy CRAFT with support for legacy traffic, a CRAFT router may allocate a certain amount of bandwidth to legacy traffic depending on its admission control policy, so clients that have not deployed CRAFT can still have access to network service. Since a user can increase his legacy traffic rate arbitrarily, a CRAFT router, in response, can use fair queuing to limit the bandwidth allocated to legacy traffic.

Let us define the CRAFT path of a flow to be the set and order of CRAFT routers traversed by the CRAFT flow. If the network path used by a CRAFT flow changes, the flow would traverse either the same CRAFT path or a different CRAFT path. If the flow traverses the same CRAFT path as the original flow, each CRAFT router can still update the TCP state emulation, and the flow can continue operating as if the route had not changed. If the flow traverses a different CRAFT path, there is at least one new CRAFT router that cannot verify the CRAFT packet header due to lack of knowledge or out-of-order fields in the header. That CRAFT router would then treat the packet as legacy traffic, and send a reset message to the corresponding sender of the flow. The sender can then re-initiate a new CRAFT flow to the destination.

5. POTENTIAL ATTACKS AND DEFENSES

In this section, we discuss several possible attacks on the basic design of CRAFT as explained in Section 4. We also propose mechanisms to defend against each attack.

5.1 Capability Forgery Attack

The size of capability affects the security and the overhead of the CRAFT system. To reduce the system overhead, we can use a short capability; for example, in our implementation of CRAFT, the short capabilities are 3-bit in length. When CRAFT uses short capabilities, an attacker can attempt to guess the capability of a packet by initiating a flow and flood the path with random capabilities. Since without a CRAFT router's key, the capability appears uniformly randomly, $1/(2^3) = 1/8$ of the attacker's guesses are expected to be accepted by the first CRAFT router on the path. If there are n deploying CRAFT router on the path between the attacker and the bottleneck link, then $1/(2^{3n}) = 1/(8^n)$ of the packets are expected to enter the bottleneck link.

In the initial phase of deployment, CRAFT routers may be only sparsely deployed, and there might not be enough CRAFT routers on the path to prevent the attacker from exhausting the bottleneck link bandwidth. For example, if only one CRAFT router is deployed between an attacker and the bottleneck link, the attacker can deny service by sending traffic at 8-times the bandwidth of the bottleneck link.

We thus propose that when a CRAFT router receives an excessive number of incorrect capabilities from a particular flow, the CRAFT router should notify the sender to switch to using long capabilities instead of the 3-bit short capabilities. We choose this approach rather than penalizing the flow to prevent an attack where the attacker alters the capability of a packet of a legitimate flow, thereby harming the performance of the benign flow.

When using long capabilities, such as those 32-bits long, the probability of guessing a valid capability is vanishingly small, and an attacker that tries to guess random capabilities will have its bandwidth reduced by a factor of 2^{32} at each legitimate CRAFT router. To switch between short capability and long capability, CRAFT packet header includes a one-bit field so that a sender and CRAFT routers can agree on using short or long capability.

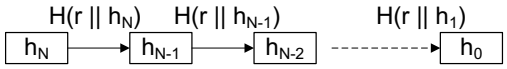


Figure 5: Generation of a hash chain

5.2 Packet ID Spoofing Attack

As mentioned in Section 4.1, a CRAFT router removes duplicate packets within a single flow that share the same Packet ID. Thus, an attacker that forges a packet with Packet ID p could cause a CRAFT router to drop the actual packet bearing Packet ID p.

To defend against this attack, we use a hash chain to authenticate the Packet ID. A hash chain consists of a series of hash values generated by a cryptographic hash function [25]. A cryptographic hash function takes a message of arbitrary length as input, and computes a fixed-length output, called a message digest. A hash function is *preimage-resistant* if given $h = H(m)$, it is computationally infeasible to obtain m.

To generate a hash chain of length N, the sender chooses two nonces, r and h_N and generates the chain by feeding back the output of a preimage-resistant hash function as the input. That is, $h_{i-1} = H(r||h_i)$, where $||$ denotes concatenation. Figure 5 illustrates how to generate a hash chain of length N. The nonce r is used to avoid reusing a hash value from previous flows.

To allow each CRAFT router to authenticate Packet IDs, a sender includes in its CRAFT connection initiation packet the values of r and h_0. Each CRAFT router stores the values r and h_0 to authenticate future hash values. When the sender sends packet with Packet ID m, the packet includes the hash value h_m in the hash field of the CRAFT header.

When a CRAFT router receives a new packet, it can authenticate the Packet ID by traversing the hash chain. For example, when the CRAFT router receives Packet ID 1 with hash element h', it will verify that $H(r||h') = h_0$, and if so, the CRAFT router will know that $h_1 = h'$. Since the cryptographic hash function is preimage-resistant, it is computationally infeasible for other users to generate h_i from h_{i-1}.

5.3 Memory Exhaustion Attack

Since a CRAFT router needs to maintain some state in memory for each flow, an attacker can attempt to exhaust the memory space of the router by establishing an excessive number of flows. CRAFT uses Portcullis [21] to mitigate this attack by allocating a per-computation-fair share of the available number of connections among all requesting end hosts.

A CRAFT router can also dynamically adjust the difficulty of the computation puzzle as a function of the amount of its free memory. Specifically, when the memory of a CRAFT router is mostly unused, the router uses easier computation puzzle to encourage more flows to use CRAFT; when the router's memory becomes low, the router uses more difficult computation puzzle to mitigate potential memory exhaustion attack.

An attacker can also attempt to exhaust the memory space of each packet. In each CRAFT packet header, a limited memory space is allocated for pre-capabilities. Thus, if an attacker fills that limited memory space with fake pre-capabilities, no CRAFT router can monitor the flow since no CRAFT router can insert its pre-capabilities into the packet header.

Since CRAFT router cannot verify that packets are received and acknowledged according to the TCP standard, a CRAFT router mitigates the packet header memory exhaustion attack by assigning flows with insufficient packet header memory to use the legacy portion of the link controlled by the CRAFT router.

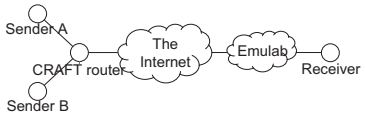


Figure 6: Internet experiment setup

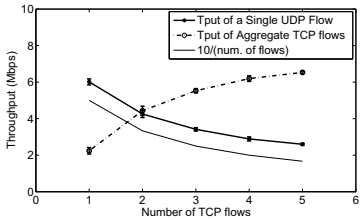


Figure 7: Fairness as evaluated by implementation. A single UDP flow, controlled by CRAFT, competes against 1–5 legacy TCP flows.

5.4 Imperfections in TCP Feedback

Because TCP congestion control measures the cwnd according to bytes acknowledged and not the total number of bytes outstanding, and because packet sizes can vary, an attacker could exploit these differences to temporarily overwhelm a link protected by CRAFT. For example, after an attacker acquires a sizable cwnd, he can send multiple 1-byte packets, each of which comes with its own IP and CRAFT header. An attacker can thus easily flood the network with traffic that is magnitudes larger than allowed by his cwnd. Many researchers have documented the effectiveness of using small payload to disrupt network service [23].

If an attacker sends multiple small-sized messages without changing his maximum segment size, we can easily stop the attack by enforcing Nagle's algorithm [20]. If an attacker sends multiple small-sized messages by changing his maximum segment size, then CRAFT defends against this attack by maintaining the maximum amount of outstanding data-plus-header. Due to space constraints, we detail our defense mechanism against this attack in Appendix B.

6. EVALUATION

In this section, we evaluate the overhead, the effectiveness, and the security of CRAFT.

6.1 Experiment Methodology

Our testbed consists of a small network in our lab which is connected over the Internet to Emulab [9], a public open network testbed. We illustrate our testbed in Figure 6. The small access network in our lab consists of two senders connected to one router, which is in turn connected to the Internet. The receiver in Emulab is also connected to the Internet. Each of our two senders sends TCP or UDP data to the receiver in Emulab over the Internet. We placed the Internet between the senders and the receiver in order to experiment with realistic cross-traffic and queuing delay. The link bandwidth between the senders and the router is 100 Mbps, and we vary the receiver's access link from 10 Mbps to 100 Mbps.

We implemented CRAFT between network layer and transport layer in Linux kernel version 2.6.20, including header insertion, processing, and extraction steps at the senders, the router, and the receiver. Our prototype CRAFT implementation uses 32-bit packet IDs, thus Packet ID, ACK ID, Reordered Packet ID, and Lost Packet ID in Algorithm 2 are all 32-bit long. Each Pre-Capability and Capability in our implementation is 3-bit long, and the timeout (TO) flag is 1-bit long.

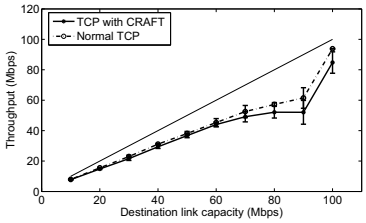


Figure 8: CRAFT packet overhead results

6.2 Fairness Measured by Implementation

We first examine how a CRAFT-compliant UDP flow shares bandwidth with other TCP flows. We let A be a group of TCP senders; and B, a single UDP sender with 10 Mbps source rate. We then let each sender in A send packets to a receiver using TCP Reno; and let B send packets using CRAFT. That is, B generates a 10 Mbps UDP traffic, but only transmits data at a CRAFT-compliant rate. All flows from A and B converge at a CRAFT router before entering the Internet and eventually reach the destination as shown in Figure 6. We implemented a user-level program to send CRAFT acknowledgments in the reverse direction in order to make a unidirectional UDP flow compatible with CRAFT.

We measure the throughput of B's CRAFT-compliant flow and the aggregate throughput of A's TCP flows while varying the number of TCP flows at A from 1 to 5. Figure 7 shows our results. The thin solid line represents the bandwidth that each flow would get if all flows enjoy the same portion of link bandwidth. The bold solid line shows the throughput of B, the CRAFT-compliant UDP flow. We observe that the CRAFT-compliant flow enjoys 20% to 55% higher throughput than its fair share.

The difference between CRAFT-compliant rate and the fair share can be attributed to TCP flows using *delayed acknowledgments*. The TCP standard recommends sending one acknowledgment for every other received packets as long as the packets meet some timing criteria. However, CRAFT limits flow rates using the maximum TCP rate, and since the TCP cwnd grows faster when the receiver acknowledges each packet instead of every other packet, CRAFT does not use delayed acknowledgments, thereby allowing a flow to gain higher throughput than the fair share.

6.3 Overhead Measured by Implementation

To enforce TCP congestion control to each flow, CRAFT incurs two types of overhead: packet header overhead and processing overhead. The packet header overhead decreases the amount of goodput with a given maximum segment size (mss) and the processing overhead increases the time a router takes to forward a packet.

Packet header overhead. A CRAFT flow stores several extra information in the packet header, thus reducing the maximum amount of data sent in each packet with a given mss. The reduced goodput can be calculated theoretically as

$$\frac{(\text{mss} - \eta_{\text{CRAFT}})}{\text{mss}} \times \text{goodput}_{\text{original}}$$

where η_{CRAFT} is the size of the CRAFT header and $\text{goodput}_{\text{original}}$ is the goodput of a flow without CRAFT deployment. Since the size of CRAFT header varies due to various optional fields, such as Reordered Packet IDs and Lost Packet IDs, we do not calculate a theoretic packet overhead, but instead determine it experimentally.

To determine the impact of the packet header overhead, we consider a TCP sender sending traffic through the Internet and Emulab to the receiver. We then compare the goodput of the TCP flow when

Table 1: Processing times for router functions

Function	Processing time
Creation of a new flow	3120ns
Lookup of flow	30ns
Calculation of hash	1720ns
Verification of capability	300ns
Update of $cwnd$	170ns
Calculation of pre-capability	610ns

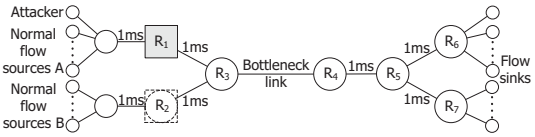


Figure 9: Simulation topology

CRAFT is in use to that of a normal TCP flow. We vary the access link bandwidth between 10 Mbps and 100 Mbps, and perform 5 runs for every data point. Figure 8 shows the mean TCP goodput for each access link rate and the 95% confidence intervals of the average. These results show that the legacy TCP flow provides 2.03% to 15.33% higher goodput than that of CRAFT.

When the access link bandwidth is under 60 Mbps, the difference between the goodputs of the two flows is not significant. We observe the biggest difference between goodputs when the access link bandwidth is 90 Mbps. The goodputs of both CRAFT and normal TCP levels off when the access link bandwidth is between 60 to 90 Mbps, we believe that this leveling behavior is caused by traffic shaping somewhere outside of our control and is irrelevant to the CRAFT overhead.

Processing overhead. Upon receiving a CRAFT packet, a CRAFT router performs several operations: looking up the flow, checking the validity of the hash, verifying the capability, updating the congestion window, and generating a new pre-capability. We use a single core of a 3 GHz Xeon process to measure the processing times required by our implementation of these functions. In particular, we use SHA256 to generate and verify the hash chain, and use RC5 to generate pre-capabilities. Table 1 shows the measured processing time for each functional block.

Our results show that a single core of a Xeon processor can handle over 300,000 packets per second. At a minimum packet size of a 20 byte IP header and 28 byte CRAFT header, this represents a bandwidth exceeding 14.4 Mbps. At a more realistic packet size of 1500 bytes, this represents 450 Mbps. Thus low bandwidth routers can use a slow processor since the processing requirement is low, and a high-bandwidth router can implement these functions on a FPGA or ASIC to process even more quickly than the Xeon CPU. These numbers are for our particular implementation of CRAFT, and might be improved in future implementations. Our measured processing times thus serve as a lower bound for CRAFT performance.

6.4 Simulation Methodology

We perform a simulation study to evaluate the effectiveness of CRAFT in providing network-wide fairness compared to other single-link protection schemes. For this comparative study, we use ns-2 simulation rather than our prototype implementation to easily implement other schemes and to modify experimental settings. We compare CRAFT against fair queueing and the flow rate estimation protocol proposed by Stoica et al. [26]. The fair queueing scheme provides a max-min allocation [5] of link bandwidth to flows. The flow rate estimation scheme calculates the time-average rate of each flow. When a flow rate is greater than its max-min fair share by a

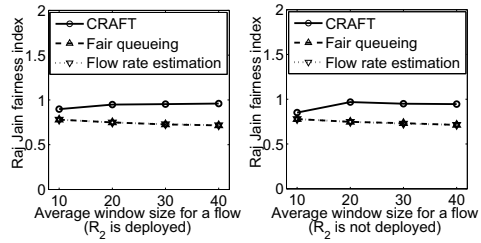


Figure 10: Comparison of fairness in terms of average window size (Attack traffic rate=10 Mbps)

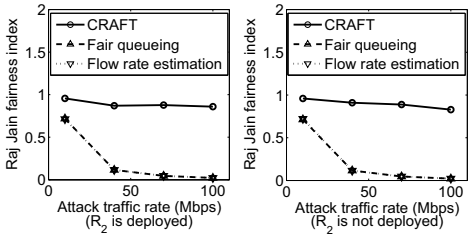


Figure 11: Comparison of fairness in terms of attack strength (Average window size=40)

threshold, the protocol flags the flow for violation. Figure 9 shows our simulation topology. Attack flow and normal flows in group A traverse a deployed router (R_1). For normal flows in group B, we examine both cases where R_2 is a deploying and non-deploying router. There is one bottleneck link that all flows share in the network. The bottleneck link is not protected by any defense mechanism and has link bandwidth 100 Mbps. There is one attacker flow which sends a CBR (Constant Bit Rate) traffic. Each group of normal flows contains 20 flows, and all normal flows use TCP Reno. In our network topology, each TCP flow experiences the same link delays. Since the RTT heavily affects the TCP throughput, we expect the average throughput of all TCP flows to be similar to each other. We measure the throughput of each flow by calculating a weighted time-average. We then calculate the Raj Jain fairness index [14]. The fairness index equation is given by $\frac{(\sum_{i=1}^{n} x_i)^2}{n\sum_{i=1}^{n} x_i^2}$ where n is the total number of flows and x_i is the throughput of flow i. The fairness index is less than or equal to 1 with equality if and only if all flows have the same throughput.

We examine the fairness provided by CRAFT, fair queueing, and flow rate estimation using two scenarios. In the first scenario, we examine the fairness with respect to the *average window size*. We define *window* to be the number of outstanding packets (packets that are sent but not yet acknowledged). Since the maximum number of outstanding packets is related to the bandwidth-delay product (BDP), we calculate the average window size to be $\frac{BDP}{S_p \times n}$, where S_p is the packet size and n is the total number of flows. To vary the BDP, we adjust the delay of the bottleneck link.

In the second scenario, we examine the fairness with respect to the constant bit rate (CBR) of the misbehaving flow. In each scenario, for each data point, we run 10 simulations and calculate the 95% confidence interval.

6.5 Simulation Results

Figure 10 compares the fairness in throughputs provided by CRAFT, fair queueing, and flow rate estimation with respect to various average window size. We let the attack traffic rate be 10 Mbps, which is a small fraction of the bottleneck link bandwidth (100 Mbps). The figure shows that CRAFT slightly outperforms other schemes

for every average window size we test. Since queue build-up happens only at the bottleneck link, the deployed routers in our tested topology do not use up the bandwidth of its immediate link. Consequently, the deployed routers do not observe congestion and cannot effectively enforce fairness. This fact results in the same fairness index for fair queueing and flow rate estimation.

Figure 11 compares the fairness in throughputs provided by CRAFT, fair queueing, and flow rate estimation with respect to the traffic rate of the attacker. Since the benign TCP-compliant flows compete against the malicious traffic, as the rate of the attacker traffic approaches the bottleneck link bandwidth, the benign flows would obtain lower throughput in the networks using fair queueing and flow rate estimation, resulting in lower Raj Jain fairness index. However, since CRAFT enforces that the attacker traffic compete with other flows in a TCP-fair manner, the Raj Jain fairness index is not significantly reduced.

In both scenarios, whether R_2 is deployed or not does not affect the resulting fairness. Moreover, CRAFT is shown to be more effective in enforcing TCP fairness in both scenarios.

6.6 Security Analysis

In this section, we show that a CRAFT router can limit all flows beyond the router to share network bandwidth in a TCP-fair manner when using short capability. TCP-fairness can be formally defined by a utility function in an optimization-based framework [18]; however, we define TCP-fairness loosely to be the behavior of the network when all flows adhere to the TCP congestion control specifications.

We first show that, when a CRAFT router is using short capability, the rate of any given flow beyond that CRAFT router must be limited by the CRAFT rate, which is defined to be the rate allocated by CRAFT to a flow given a loss pattern in a network. We then argue that the CRAFT rate is TCP-fair. That is, a CRAFT flow must adhere to the TCP congestion control algorithm.

To show that the rate of a flow is limited by the CRAFT rate, we observe that a CRAFT router disregards all packets with invalid capabilities. That is, if a flow wishes to increase its rate above the CRAFT rate, it needs to send more capabilities than it receives. However, an attacker cannot consistently predict future acknowledgments, and too many erroneous attempts will cause the CRAFT router to force the attacker to switch to using long capabilities, thus further reducing the success rate of guessing. Moreover, since the rate can only monotonically decrease as the flow traverses more CRAFT routers, a single CRAFT router (namely the closest one to the sender) is able to limit the flow rate below CRAFT rate.

When a CRAFT flow uses long capability, an attacker could potentially argue that someone else is trying to spoof the capability, thereby evading any penalties and CRAFT rate limit. However, with a 32-bit long capability, only one in 2^{32} excess packets carries a valid capability. To put this in perspective, an attacker flooding an OC-768 link at line-speed can only squeeze 9.3 bps past the first CRAFT router. Each additional CRAFT router provides a further reduction in bandwidth.

We now argue that CRAFT flows and regular TCP flows could share a link with TCP-fairness on a legacy link. It is immediately obvious that a TCP flow would share the link with a CRAFT flow in a TCP-fair manner. That is, without considering what the CRAFT flow does, the TCP flow adheres to TCP congestion control. In the other direction, without considering the TCP flow, the CRAFT flow is limited by its CRAFT rate, which is in turn determined using the TCP congestion control standard. That is, the CRAFT flow must also be TCP-fair. Therefore, CRAFT flows and other TCP flows are able to share a legacy link in a TCP-fair manner.

7. RELATED WORK

In this section, we overview related work on monitoring misbehaving flow, misbehaving TCP receiver and capability-based systems.

Monitoring misbehaving flow. Floyd et al. [11] use a TCP throughput equation to determine the proper throughput of a flow. They then propose punishing a flow that uses higher throughput than that calculated throughput. A basic assumption of their work is that a sender does not cheat on parameters in the TCP throughput equation. However, an attacker can break this assumption. For example, the TCP throughput is reversely proportional to the round trip time. Hence, an attacker can intentionally delay its packets so that the attacker can enlarge the throughput calculated by the router. This problem originates from the *weak trust model* of their method. Stoica et al. [26] estimate the rate of a flow and use the estimated rate to distribute fair share of a link. Unlike the approach proposed by Floyd et al., the approach proposed by Stoica et al. does not place any trust in a sender. However, due to the statistical nature of estimating a rate, false detection can occur and a short but heavy flow can greedily occupy a portion of bandwidth temporarily.

Misbehaving TCP receiver. Ely et al. [8] use a random nonce to defend against a receiver selfishly exploiting ECN. Savage et al. [22] considered a misbehaving TCP receiver that acknowledges several pieces of a received packet (ACK division), acknowledges one received packet several times (dup-ACK spoofing), or acknowledges future packets in advance (opt-ACK). Savage et al. proposed a modified TCP protocol to defend against untrusted receiver's misbehavior. Sherwood et al. [23] demonstrated that a misbehaving TCP opt-ACK receiver together with a small maximum-segment-size can induce the sender to inject a large amount of data into the network. These works assume that a sender is benign and trusted. Our work assumes that both of the sender and the receiver of a flow can be colluding attackers.

Capability-based systems. Capability-based systems enable receivers to select which sender can send traffic to them. Anderson et al. suggest capabilities to defend against bandwidth exhaustion attacks [2]. Yaar et al. proposed the SIFF protocol [28], and Yang et al. proposed the TVA protocol [29] that are concrete designs of capability-based DoS defense systems. Liu et al. compared the performance of various capability-based systems and filter-based systems [17]. Parno et al. proposed the Portcullis protocol based on computation puzzles. By prioritizing service requests based on the computational power of the clients, Portcullis can prevent an attacker from flooding links with connection initialization packets to exhaust the capability request channel [21]. CRAFT can be considered as a capability-based mechanism to secure congestion control protocols.

8. CONCLUSION

We have presented the CRAFT protocol to provide a high level of security against misbehaving users, by enforcing TCP-fairness on all flows that have traversed a CRAFT router. The central goal of CRAFT is to achieve a system where a CRAFT-enabled router can prevent a flow from causing unfairness on any downstream link, even if the link is surrounded by legacy routers.

A single CRAFT router can protect all links behind it; thus CRAFT can be deployed with little cost. Our experiments show that CRAFT enforces TCP fairness while incurring little overhead in packet header size and computation time. Our simulations show that in realistic partial-deployment environments, CRAFT can provide superior fairness than previously suggested mechanisms.

9. REFERENCES

[1] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. RFC 2581, April 1999.

[2] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial-of-Service with capabilities. In *Proceedings of SIGCOMM '04*, volume 34, pages 39–44, New York, NY, USA, 2004. ACM.

[3] H. Balakrishnan, V.N. Padmanabhan, S. Seshan, and R.H. Katz. A comparison of mechanisms for improving tcp performance over wireless links. *Networking, IEEE/ACM Transactions on*, 5(6):756 –769, Dec 1997.

[4] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of CCS '93*, pages 62–73, New York, NY, USA, 1993. ACM.

[5] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1991.

[6] L. S. Brakmo and L. L. Peterson. TCP Vegas: end to end congestion avoidance on a global Internet. *Selected Areas in Communications, IEEE Journal on*, 13(8):1465–1480, Oct 1995.

[7] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *In Proceedings of SIGCOMM '89*, pages 1–12, New York, NY, USA, 1989. ACM.

[8] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust congestion signaling. In *Proceedings of ICNP '01*, pages 332–341, Washington, DC, USA, 2001. IEEE Computer Society.

[9] Emulab (http://www.emulab.org).

[10] S. Floyd. Highspeed tcp for large congestion windows. RFC 3649, December 2003.

[11] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *Networking, IEEE/ACM Transactions on*, 7(4):458 –472, Aug 1999.

[12] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on*, 1(4):397–413, 1993.

[13] Y. He, M. Faloutsos, S. Krishnamurthy, and B. Huffaker. On routing asymmetry in the Internet. In *Proceedings of GLOBECOM '05*, volume 2, November 2005.

[14] R. Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991.

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

[16] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, February 1997.

[17] X. Liu, X. Yang, and Y. Lu. To filter or to authorize: network-layer DoS defense against multimillion-node botnets. In *Proceedings of SIGCOMM '08*, pages 195–206, New York, NY, USA, 2008. ACM.

[18] S. H. Low. A duality model of TCP and queue management algorithms. *Networking, IEEE/ACM Transactions on*, 11(4):525–536, Aug. 2003.

[19] S. H. Low and D. E. Lapsley. Optimization flow control. i. basic algorithm and convergence. *Networking, IEEE/ACM Transactions on*, 7(6):861 –874, Dec. 1999.

[20] J. Nagle. Congestion control in IP/TCP Internetworks. RFC 896, January 1984.

[21] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu. Portcullis: protecting connection setup from Denial-of-Capability attacks. In *Proceedings of SIGCOMM '07*, volume 37, pages 289–300, New York, NY, USA, 2007. ACM.

[22] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. Tcp congestion control with a misbehaving receiver. *SIGCOMM CCR*, 29(5):71–78, 1999.

[23] Rob Sherwood, Bobby Bhattacharjee, and Ryan Braud. Misbehaving tcp receivers can cause internet-wide congestion collapse. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 383–392, New York, NY, USA, 2005. ACM.

[24] W. R. Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[25] D. R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC, 2002.

[26] I. Stoica, H. Zhang, and S. Shenker. Self-verifying csfq. In *Proceedings of INFOCOM '02*, volume 1, pages 21 – 30 vol.1, 2002.

[27] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *Proceedings of INFOCOM '04*, volume 4, pages 2514 – 2524 vol.4, March 2004.

[28] A. Yaar, A. Perrig, and D. Song. Siff: a stateless Internet flow filter to mitigate DDoS flooding attacks. In *Proceedings of S & P '04*, pages 130–143, May 2004.

[29] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *Proceedings of SIGCOMM '05*, volume 35, pages 241–252, New York, NY, USA, 2005. ACM.

APPENDIX

A. BACKGROUND ON TCP

In this section, we introduce some concepts and acronyms related to TCP. The purpose of this section is not to explain all the details of TCP but to briefly introduce the main concepts of TCP congestion control. Readers who would like more details on TCP can refer to the literatures [1, 24]. The details provided in this section are based on standard [1].

In TCP, each byte is given a contiguous *sequence number* for each direction of a TCP connection. Each packet contains the sequence number of the first byte contained in that packet, as well as an *acknowledgement number*, which represents the sequence number of the last contiguously received byte for the opposite-direction traffic.

TCP congestion control treats the network as a black box, observing the end-to-end delay and loss characteristics and choosing a data sending rate compatible with all intermediate links. The main goal of the TCP congestion control is to estimate the available network bandwidth and adjust the offered load accordingly. A TCP sender adjusts its offered load by changing its *congestion window* (cwnd), which represents the maximum number of unacknowledged data bytes that the sender releases into the network at a time.

When an acknowledgment packet reaches the sender, the sender increases its cwnd. There are two operation phases in which a sender increases its cwnd: *slow start* and *congestion avoidance*.

In the slow start phase, a TCP sender increases its cwnd rapidly since it assumes that a network is much under-utilized. When the sender's cwnd reaches a threshold (*ssthresh*), it enters the congestion avoidance phase. In the congestion avoidance phase, a TCP sender assumes that the network is close to being congested. Hence, the sender increases its cwnd slower than the sender would in the slow start phase.

If the increased cwnd caused the network to be congested, packets will be lost in the congested links in the network. TCP detects the congestion through packet losss, and in turn, detect packet losses by using duplicate acknowledgments. When a packet is lost, the receiver reacts to an unacknowledged out-of-order packet (that is, one with sequence number greater than the next contiguous sequence number) by sending a *duplicate acknowledgment*, which has the same acknowledgment number as the previous acknowledgment packet. When the sender receives a third duplicate acknowledgment, it concludes the packet after the acknowledged packet to have been lost, and retransmits that packet in a process known as *fast retransmit*. Once the retransmitted packet is acknowledged, the sender goes through *fast recovery* by halving the size of its cwnd, and setting the ssthresh to the new cwnd, and then the sender enters congestion avoidance again. When a large number of packets are lost at once, a sender may not receive any duplicate acks. In this case, the sender times out, reset its cwnd and ssthresh, and starts from the slow start phase again.

B. IMPERFECTIONS IN TCP FEEDBACK

In Section 5.4, we noted an attack where the attacker changes his packet size and use packet header to amplify his attack traffic. In order to avoid such an attack, CRAFT enforces certain protocols that TCP does not require. Namely, CRAFT enforces Nagle's algorithm [20]; that is, a CRAFT router allows each flow to have one outstanding packet that is of size less than the maximum segment size. By enforcing Nagle's algorithm, we can prevent an attacker from sending multiple small sized packets.

When an application needs to send a large number of small packets, it may decide to reduce its maximum segment size for a period of time. In CRAFT, the sender notifies all routers each time it changes the maximum segment size. When a node increases its maximum segment size, the cwnd can remain unchanged, because the total number of bytes that can be sent for a given cwnd decreases as the maximum segment size increases. This is because as the maximum segment size increases, the number of packets that can be sent for a given cwnd decreases, thus the amount of overhead associated with those packets also decreases.

Conversely, when decreasing the maximum segment size, the amount of bandwidth associated with a given cwnd increases. If we allow the sender to decrease its maximum segment size without also decreasing its cwnd, then the attacker can temporarily send at a rate much greater than its fair rate. Thus, when a node *decreases* its maximum segment size, we adjust its cwnd to ensure that its old maximum data-plus-header in flight is equal to its new maximum data-plus-header in flight, by satisfying the expression

$$\text{cwnd}_{\text{old}} + \eta\frac{\text{cwnd}_{\text{old}}}{\text{SMSS}_{\text{old}}} = \text{cwnd}_{\text{new}} + \eta\frac{\text{cwnd}_{\text{new}}}{\text{SMSS}_{\text{new}}},$$

where SMSS is the sender's maximum segment size, and η is the size of the overhead, including the TCP header and the CRAFT header.

C. DISCUSSION

C.1 Various flavors of TCP

As introduced in Section 2.2, we handle various flavors of TCP by emulating the TCP congestion control specified in RFC 2581 [1] and increase the emulated cwnd in the most aggressive manner. Several congestion controls and bandwidth allocation mechanisms do not follow the TCP congestion control stated in RFC 2581. As such, current version of CRAFT cannot accurately emulate the state of these flows and might not support flows using these congestion control algorithms. These TCP variants behave differently from RFC 2581 in two respects: congestion determination and window evolution. For example, TCP Vegas [6] uses increase in measured delay as a sign of congestion instead of the triple duplicate acknowledgments used by RFC 2581. Also, BIC TCP [27] updates the congestion window differently from the protocol specified in RFC 2581. Though these variants are not currently supported by CRAFT, as long as these variants are still TCP-friendly, future versions of CRAFT could allow for user selection of multiple state-tracking strategies.

C.2 Deployment Model

Routers. We have shown in Section 6.6 that a CRAFT router can protect all downstream links from upstream traffic. That is, a network provider can protect all its internal links from outsiders by upgrading only its border routers. Moreover, if a service provider trusts one of its neighbors less than others, the service provider can first deploy CRAFT at the border router with the neighbor that is less trusted. A provider needs not deploy CRAFT at all routers at once; instead, a provider can slowly phase out its legacy links. CRAFT thus allows a provider to avoid both expensive one-time cost and complicated simultaneous deployment.

End-Hosts. Moreover, the service provider can lessen the end users' burden of software upgrade by installing CRAFT proxies in the customer's access network. The proxies then can process and forward all capabilities and pre-capabilities on the users' behalf.

Applications. Applications that use a non-TCP transport protocol can still be made compliant with CRAFT. When a UDP flow wants to send at a rate less than its CRAFT rate, all packets associated with that flow are sent as CRAFT packets. When such UDP flow wants to exceed its CRAFT rate, it selects the less important packets and sends them as best-effort (i.e. using the legacy portion of CRAFT links), so that the packets that are important are sent with higher priority at no more than the CRAFT rate.