A Middlebox-Cooperative TCP for a non End-to-End Internet

Ryan Craven
Naval Postgraduate School
rcraven@nps.edu

Robert Beverly
Naval Postgraduate School
rbeverly@nps.edu

Mark Allman
ICSI
mallman@icir.org

ABSTRACT

Understanding, measuring, and debugging IP networks, particularly across administrative domains, is challenging. One particularly daunting aspect of the challenge is the presence of transparent middleboxes—which are now common in today's Internet. In-path middleboxes that modify packet headers are typically transparent to a TCP, yet can impact end-to-end performance or cause blackholes. We develop TCP HICCUPS to reveal packet header manipulation to both endpoints of a TCP connection. HICCUPS permits endpoints to cooperate with currently opaque middleboxes without prior knowledge of their behavior. For example, with visibility into end-to-end behavior, a TCP can selectively enable or disable performance enhancing options. This cooperation enables protocol innovation by allowing new IP or TCP functionality (e.g., ECN, SACK, Multipath TCP, Tcpcrypt) to be deployed without fear of such functionality being misconstrued, modified, or blocked along a path. HICCUPS is incrementally deployable and introduces no new options. We implement and deploy TCP HICCUPS across thousands of disparate Internet paths, highlighting the breadth and scope of subtle and hard to detect middlebox behaviors encountered. We then show how path diagnostic capabilities provided by HICCUPS can benefit applications and the network.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*TCP*; C.4 [**Computer-Communication Networks**]: Performance of Systems—*Measurement techniques*

Keywords

TCP; Middlebox; Header Integrity; Header Modifications

1. INTRODUCTION

The traditional Internet architecture envisions intelligence at the ends and simplicity in the middle [13]. This traditional view, where the network focuses on forwarding packets, is long gone. Middleboxes now actively interpose on communication for a multitude of reasons [9], including implementing acceptable use policies, maintaining regulatory compliance, thwarting attacks, censoring or monitoring users, expanding address space, limiting or balancing resources, and generating revenue. However, the functional consequences of middlebox mechanisms, which are frequently decoupled from the end-to-end path, may be both intentional and unintentional. The prevalence of middleboxes, and the wide variety of behaviors they exhibit, is well-established by previous empirical research [18, 31, 36, 40, 46].

One side effect of middleboxes is that they make the task of debugging networks—already a difficult problem, especially across administrative domains—even harder by introducing a variety of unknowns [31]. Because of their privileged position in the network, it is important that middleboxes not adversely impact (e.g., block or degrade) the traffic of systems or users outside of their intended scope.

Unfortunately, middleboxes have been shown to induce not only the intended changes in traffic behavior, but also unintended side effects. Legacy equipment, non-standard implementations, and misconfigurations are known to interact with middleboxes to mutate critical packet fields, destroy semantics, create unintended protocol interactions, and violate the end-to-end nature of the Internet. For instance, previous measurements have shown that middleboxes frequently misconstrue and block new IP or transport functionality [6, 19, 25]. Thus, an important and often under-appreciated class of network problems are the result of *non-malicious* and *unintentional* middlebox behavior.

While clean-slate designs (e.g., [45]) and software-defined management (e.g., [38, 41, 42]) attempt to more cohesively integrate middleboxes into the network, they depend on deployment and use; TCPs in the wild must continue to contend with a variety of middlebox behaviors. In contrast, we advocate for empowering TCP endpoints with awareness of middlebox packet header modifications along a path. Similar to how TCP currently infers end-to-end congestion state, a TCP host with knowledge of the end-to-end packet header modification state can better match its behavior to the capabilities of the path. By cooperatively adapting to middleboxes, TCP can improve performance. Perhaps more importantly, endpoints can realize the benefits of protocol innovation as new TCP or IP functionality can be more safely deployed and enabled in routers and operating systems.

We implement and deploy TCP HICCUPS (Handshake-based Integrity Check of Critical Underlying Protocol Se-

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.
Copyright 2014 ACM 978-1-4503-2836-4/14/08 ...$15.00.
http://dx.doi.org/10.1145/2619239.2626321

mantics). HICCUPS permits endpoints to cooperate with currently opaque middleboxes without prior knowledge of their behavior. HICCUPS is incrementally deployable, backward compatible, introduces no new IP or TCP options, and adheres to all TCP/IP standards, i.e., will traverse the same paths as traditional TCP. HICCUPS provides bidirectional in-band measurement and feedback such that a TCP sender can infer the state of how her packet headers were received by the other end of the connection. With widespread deployment, HICCUPS would also enable a new general path diagnostic capability in the same way that `ping` (ICMP echo) can be used to test paths without prior endpoint coordination. We make the following primary contributions:

1. Design of TCP HICCUPS, an incrementally deployable improvement on TCP to reveal packet header manipulation to both ends of a TCP connection.
2. Real-world implementation and testing of TCP HICCUPS in the Linux kernel.
3. Deployment of, and measurements from, TCP HICCUPS across thousands of disparate Internet paths.
4. Demonstrable instances of degenerate middlebox behavior and the ways in which HICCUPS cooperation improves transfer performance.

2. BACKGROUND

Some Internet packet headers were designed to experience modification, for instance the IP time-to-live and checksum are decremented and recomputed respectively at each hop. Other fields such as the Differentiated Services Code Point (DSCP) [37] only have significance within each transiting network, having no guarantee to be constant along a path.

However, other fields have *end-to-end significance*, for example: source and destination addresses, transport ports, TCP flags, flow control window, and TCP options. Modifications to fields intended for interpretation only by endpoints can lead to subtle and unintentional problems. In the worst case, traffic can be blocked. In other instances, performance can suffer—sometimes dramatically. In this section, we first discuss some of the impacts resulting from the current environment of opaque middleboxes. We then examine prior work in providing integrity and diagnostics of packet manipulation. Lastly, we examine emerging research toward middlebox cooperation.

2.1 TCP/IP Misinterpretation

The unintended effects and architectural issues of middleboxes are well-documented. Medina et al. cataloged issues stemming from unexpected middlebox interactions [36]. Different behaviors were observed depending on the use of IP or TCP options and Explicit Congestion Notification (ECN).

Measurements from Honda et al. [25] discovered paths with middleboxes that strip both known and unknown TCP options, perform sequence numbers translation, and even exhibit port-dependent behavior, e.g., options removed from packets destined to a random transport port, but not port 80. At least 25% of the paths tested traversed a middlebox whose behavior depended on the packet's transport-layer. Not only is such interference detrimental to the validity of the protocol interactions, it is difficult to diagnose—making troubleshooting a complex endeavor.

We focus on *unintentional* and *unintended* middlebox behaviors. Several examples we find in the wild include:

• ***Sequence Number Translation:*** To mitigate security issues inherent in predictable TCP sequence numbers, some paths contain network elements that randomize and remap sequence numbers on behalf of a host [12] (the assumption being that hosts cannot be trusted to perform proper randomization). While investigating a performance problem at our own institution, we found that while sequence numbers were being remapped in the standard TCP header, they were not being remapped in SACK blocks—which appear in the options portion of the header. This renders selective acknowledgment information useless, impacting bulk transfer performance. Diagnosing this subtle error required trained engineers using cooperating endpoints.

• ***Options:*** TCP options—which convey information between endpoints that is not germane to the network itself—are frequently deleted, added, or modified, disrupting various protocol extensions. For example, some paths add a Maximum Segment Size (MSS) if not present, or rewrite MSS, impacting performance if the true path MSS is larger or smaller. Other paths modify or remove the Window Scaling option, causing a remote endpoint to misinterpret the receiver window and incorrectly apply flow control. Not only do these common options experience modification, newer options are often stripped or blocked. For example, legacy middleboxes that are unaware of Multipath TCP [20] may strip those options, impacting performance.

• ***Type of Service:*** The original IPv4 specification includes a byte for "type-of-service." That byte has long since been redefined to consist of two bits for Explicit Congestion Notification (ECN) and six bits of DSCP. Yet, a non-trivial fraction of devices and paths still use the previous definition and rewrite or zero the entire byte. This rewriting can prevent a TCP connection from utilizing router congestion signals, or more seriously, cause a TCP connection to falsely interpret congestion [6]. Managing congestion and improving TCP performance are critical to content providers and data centers. As one large content provider stated: "we want to enable ECN, but do not because enabling ECN may adversely affect some of our users." [3]

Such behavior by middleboxes can make it a challenge to diagnose the cause of various performance and connectivity issues. Even more troubling is the unintended effect middleboxes can have on protocol innovation and adoption: any new option, repurposed field, or otherwise unrecognized behavior is often misunderstood or blocked [19, 25].

2.2 Integrity

Most integrity mechanisms built into Internet communication protocols are intended solely for error detection, such as CRC and Internet checksums [43]. Such checksums must always match integrity for a packet to be accepted, lest a device assume the packet experienced some transmission error. By necessity, any middleboxes modifying the header must also recompute any error detection checksums.

A natural response to the middlebox-induced cases of misinterpretation cited above is to employ tamper-resistant mechanisms (i.e. cryptography to encrypt and sign traffic) to prevent alterations during transmission. Such mechanisms have been developed for the network [28, 29], transport [8, 32, 44] and application [21] layers. At the application layer, the use of encryption to protect payloads has been well-adopted and is pervasive throughout the Internet. However, at the lower layers the problems of key sharing between anonymous hosts hinder adoption while imposing unnecessary cost on higher layers. Furthermore, at lower layers users often desire properly functioning middlebox intervention—e.g., to share a single public IP address among multiple devices in their home—and therefore have a disincentive to utilize tamper-resistance. As we show in §3, we relax the requirements for tamper-resistance to implement a tamper-evident design that is more cooperative with modern middleboxes.

Tracebox is a diagnostic tool to detect changes made by middleboxes along the forward path [17]. Using a similar methodology as traceroute, tracebox progressively increments the TTL of packets while additionally inferring the presence of middleboxes by comparing ICMP time exceeded quotations [5] with the originally sent packets. One drawback to the method are the inconsistencies involved with ICMP router quotations [34]. Even though the approach works for a majority of paths—Detal et al. find that $\approx$80% of the paths they examined contained at least one full-quoting router—the paths that the method cannot test likely contain the most legacy equipment that could impact TCP.

While HICCUPS and tracebox share a common goal, there are key differences between the two approaches. Tracebox is a measurement tool that relies on the network to produce and respond with diagnostic feedback. Whereas HICCUPS is in-band, tracebox requires extra diagnostic packets, unblocked ICMP, and router response. Further, HICCUPS is tightly integrated into TCP, understands both the forward and reverse path, and allows TCP to make inferences about whether it is being misinterpreted.

2.3 Middlebox Cooperation

Middleboxes are an Internet reality. The middlebox market, estimated to reach more than $10B by 2016 [1], is ample evidence that middleboxes provide value. With the prevalence and reach of middleboxes increasing, several approaches seek explicit accommodation.

Walfish et al. propose a new architecture that gives all entities globally unique identifiers in a flat namespace while allowing for explicit intermediate packet processing [45]. More recent research seeks to reduce the sprawl of standalone, non-cohesive middleboxes and employ new software-defined approaches so they can be more easily managed [22, 38, 41, 42]. Meanwhile, multiple vendors have recognized the problem of middlebox cooperation and have added TCP options that allow middleboxes along a path to voluntarily participate in their transparent discovery [30]. While these schemes make it easier to manage middlebox deployments and keep them up-to-date, they depend on adoption and use. Software-defined management, for example, is confined to single administrative domains. However, TCP hosts in the wild must contend with a wide variety of middleboxes.

Each of the above schemes require some form of active cooperation by middleboxes or their operators. We emphasize that this is different from the manner in which HICCUPS is cooperative with middleboxes. We have designed HICCUPS so that it does not interfere with middlebox operation. It does not require active participation by middleboxes.

3. DESIGN SPACE

We aim to identify and address the class of problems involving misconfigured, non-standards conforming, and legacy in-path middleboxes impacting normal traffic behavior. In the same way that TCP currently infers end-to-end congestion state, a TCP instance aware of the end-to-end packet header modification state can better match its behavior to the capabilities of the path. For instance, TCP could improve path performance by selectively enabling or disabling extensions (e.g., ECN, SACK, Multipath TCP, etc.) when they are at risk of being misinterpreted on a given path.

At a high-level, we desire a TCP-based integrity check to detect in-network packet header modifications. Such modifications today are opaque, e.g., Medina [36] could not disambiguate between "a middlebox stripping or mangling the option or the web server not supporting [the option]"—our design must provide such visibility.

The space of possible solutions is large. While seemingly straightforward, no prior work accommodates all of the properties and functionality we require:

- ***In-band:*** Many paths block out-of-band traffic (e.g., ICMP) or treat it differently. By having both the detection and feedback mechanisms in-band, we hope to maximize the detection rate.
- ***Lightweight:*** The design should result in a minimal amount of overhead in terms of computation, communication, and RTTs.
- ***Symmetric feedback:*** It is important that hosts at each end of a connection know if and how their packets were modified in flight.
- ***Incrementally deployable:*** The design should not interfere with endpoints that have not been upgraded, nor require any updates to in-network elements.
- ***Improves TCP:*** The design should endow TCP endpoints with the information needed to reason about how the options and extensions they employ are interpreted by the remote endpoint.
- ***Middlebox-cooperative:*** The design should not impede or circumvent properly functioning middleboxes, and not exacerbate degenerate middlebox behaviors.
- ***End-to-end:*** Paths exhibiting modifications are the same paths most likely to block or strip any new instrumentation. Values should be properly communicated end-to-end.
- ***Granularity:***: Endpoints should be able to determine which packet header fields were changed.

In addition, our design should not enable any new attacks on the system (e.g., amplification, spoofing, flooding, etc.)

3.1 Meeting our Architectural Objectives

How well does our TCP HICCUPS (detailed in §4) meet the aforementioned requirements? We show the degree to which HICCUPS and other relevant prior works from §2 provide such functionality in Table 1. In particular, note that HICCUPS represents a unique point in the design space.

A key insight enabling our solution is the fresh point of view afforded by our security model of the inadvertent adversary, a non-malicious system in the middle of a connection that is corrupting critical packet semantics. Much of the prior research has focused on the edges of the spectrum: protecting integrity from either transmission errors or from strong adversaries (§2.2). When operating under the model of the misconfiguration adversary, such solutions either fail to expose problematic behaviors or make too many sacrifices in pursuit of strong cryptographic assurances.

Table 1: HICCUPS in the context of existing and proposed integrity and middlebox cooperation schemes (● indicates that a scheme fully meets the criterion; ○ indicates a scheme does not meet the criterion).

Scheme	In-band	Light-weight	Symmetric feedback	Incrementally deployable	Improves TCP	Middlebox-cooperative	End-to-end	Granularity
Checksums [43]	●	●	○	●	○	●	◔	○
Tcpcrypt [8]	●	○	◑	◕	○	◔	◕	○
Tracebox [17]	○	◑	◑	◕	○	◕	◑	◕
SIMPLE [38]	○	◑	○	◔	◕	●	◔	●
HICCUPS	●	●	●	●	●	●	●	●

For example, standard checksums require middleboxes to recalculate them after changes and also provide no method to expose results of the integrity exchange to the endpoint initiating the connection. The lack of an explicit notification back to the sender when its packet arrives with a bad checksum has been a previously noted weak point [43].

HICCUPS allows both endpoints of a connection to receive feedback about the integrity of the (potentially asymmetric) paths taken by their traffic. Working within TCP makes checking bidirectional path integrity easier since the notion of a conversation is already clearly defined. By equipping the headers of the TCP 3WHS with integrity, we hope to capture the majority of performance-impacting modifications by middleboxes. While some issues with extensions would require covering the full connection to explicitly detect, protecting even just the 3WHS presents a large step toward making inferences that can improve performance. We present designs for protecting the full connection in [16].

Once a system is HICCUPS-enabled, it can perform integrity checks with other HICCUPS-enabled systems through any open remote TCP port. HICCUPS TCP stacks are interoperable with non-HICCUPS TCP stacks and its traffic appears no different to network devices from typical TCP traffic. If widely deployed, HICCUPS would provide a general diagnostic mechanism in a manner similar to `ping` and `traceroute`, wherein explicit endpoint cooperation is not required to measure a path. This "always on" property implies that utility to TCP will increase with the deployment of HICCUPS. Any HICCUPS-enabled system with an open service, e.g., a web server with TCP port 80 open, will support a HICCUPS measurement.

In contrast to `ping` and `traceroute` (as well as other middlebox detection tools like tracebox), we do not leverage out-of-band mechanisms (e.g., ICMP) in our design so as to avoid complications inherent in relying on external dependencies. In particular, note the difficulties with Path MTU Discovery as TCP operation is linked with ICMP traversal [33, 36]. A new method of PMTUD was later written that did not rely on receipt of the ICMP Packet Too Big (PTB) messages [35].

4. TCP HICCUPS

As a real-world instantiation of our architectural objectives, we develop the Handshake-based Integrity Check of Critical Underlying Protocol Semantics (HICCUPS), an enhancement to TCP. HICCUPS can assist TCP in determining the most appropriate set of end-to-end parameters that best fit the middleboxes along a particular path. In particular, HICCUPS would allow a TCP instance to reason about how the options and extensions it employs are interpreted by a remote endpoint, and subsequently make inferences about when it is safe to make use of new extensions. HICCUPS benefits TCP in two primary ways:

1. Equips TCP with critical path information that would allow it to more safely increase the use of performance-enhancing extensions relative to ultra conservative approaches where new extensions are disabled by default or left to run in "server-mode" *à la* ECN as deployed and configured in modern operating systems[1].

   Examples: ECN, Multipath TCP
2. Provides early warning of potential middlebox-induced issues with an extension that is enabled by default. TCP could proactively disable or ignore the extension to improve performance.

   Examples: SACK, Window Scaling

Our solution helps enable these performance benefits by monitoring the state of packet headers through an in-path integrity exchange, essentially creating a lightweight *tamper-evident* seal across the headers. The results of the exchange allow endhosts to work within the current path conditions to tailor the set of extensions they use to the middleboxes in the path between them.

4.1 Overview

Working within TCP to enable detection of in-path header modifications while maintaining interoperability with current network infrastructure and endhosts is a difficult systems problem. We first provide an overview of HICCUPS:

1. HICCUPS transmits packet header integrity information by *overloading* three header fields of the TCP 3-way handshake that can contain a flexible value: initial sequence numbers, initial IPIDs, and initial flow control windows. Doing so yields the highest degree of interoperability with the widest number of paths, but places tight constraints on the amount of information transmitted. See §4.2 for more.

2. When HICCUPS places integrity information in the sequence number, randomness is added for spoofing protection. See §4.2 for more.

3. The integrity information transmitted by HICCUPS includes three 12-bit hash fragments, each communicated through one of the overloaded fields in item 1. Spreading integrity across fields provides resilience to a single modification affecting any one of the three fields, e.g., sequence number translation. See §4.3 for more.

4. Reverse path integrity includes status values that enable a HICCUPS host to discover when modifications occur to just the forward path, just the reverse path, or to both paths. See §4.3 for more.

[1]In server-mode ECN, a TCP endpoint will not initiate ECN, but will negotiate ECN if initiated by the client.

5. HICCUPS supports granularity in its integrity checks. A set of coverage types allows endhosts to dynamically specify subsets of fields to be protected by HICCUPS. (§4.4)

6. As an additional protection, e.g., against middleboxes that might, in the future, actively attempt evasion, HICCUPS enables applications to optionally protect integrity with an *ephemeral secret* (§4.5). This secret limits false inferences of integrity in the event that a change is made and the integrity is recomputed. §4.6 provides a discussion of how we extend the Linux socket API to provide this feature.

4.2 Overloading Header Fields

To minimize interference from legacy and non-standard middleboxes, we avoid either redefining any field semantics or using any new IP or TCP options. New options and/or new semantics exacerbate middlebox incompatibility and we want to avoid being subject to the same issues we wish to detect. Furthermore, the TCP option space is already overcrowded [25] with many well-established extensions. By not competing for new space, we hope to avoid unintended interactions and facilitate easier adoption.

In order to integrate the integrity check within TCP/IP, we *overload* three specific fields in the headers that are allowed a certain degree of flexibility: the TCP initial sequence number (ISN), the initial IP Identification field (IPID), and the initial TCP flow control window (RCVWIN)[2]. Each end of the connection chooses its own 32-bit ISN, 16-bit IPID, and 16-bit RCVWIN resulting in a total of 64 bits at each end of the connection to be used by HICCUPS.

While HICCUPS adds meaning to the ISN, the ISN must remain unpredictable to thwart spoofing and off-path packet injection attacks. We therefore add randomness to our ISN integrity function. The bits of randomness, or salt, are sent in the clear to allow the remote host to verify the integrity. We place the random salt value in the lower half of the ISN and exclusive or (XOR)-encode the the integrity information in the upper half of the ISN with the same salt value.

Since the new ISN is created using a function of packet data, it will not be fully random, i.e., the probability of an off-path attacker being able to correctly guess the ISN is greater than 2^{-32}. In the extreme worst case, the probability is 2^{-16}, but that requires an attacker know: the flow tuple including the ephemeral port [2], the coverage type used (§4.4), and the exact contents of any packet header fields covered by that type. In practical use, an off-path adversary will not know the coverage type—two of which also cover the ephemeral port.

4.3 Integrity Exchange

Fundamental to HICCUPS is exchanging integrity and communication of the check results. Given a safe and reliable transmission mechanism (§4.2), we are able to exchange integrity, coverage, and status. Our objective is to utilize the 64 bits at our disposal in such a way as to be robust against paths that corrupt any of the three integrity exchange fields. In order to withstand a change to any single overloaded field, we place a portion of the integrity information, along with a copy of the coverage or status, in each of the three fields.

Figure 1 presents a simplified timing diagram illustrating the exchange of integrity between two HICCUPS-enabled

[2]Other works leverage these fields for steganographic covert channels [14]. In contrast, our goal is fundamentally different: the HICCUPS algorithm and field population is public.

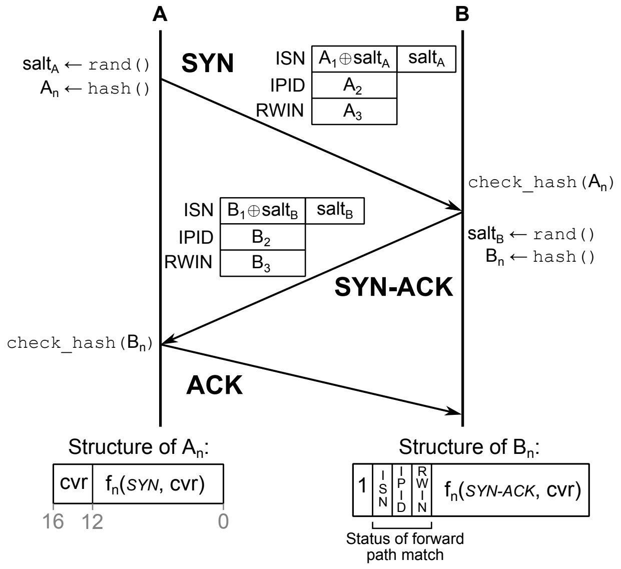

Figure 1: HICCUPS integrity exchange: A's `SYN` overloads random fields with integrity and coverage flags. B's `SYN-ACK` encodes reverse path integrity and forward path status.

hosts, A and B. Unless otherwise noted, HICCUPS follows the TCP standard and uses standard congestion control algorithms (e.g., our implementation retains Linux CUBIC behavior). Host A initiates the active open with B. Both SYNs of the three-way handshake (3WHS) utilize the ISN, IPID, and RCVWIN fields to transmit up to 16 bits each of integrity information, denoted in the figure as A_n and B_n where $n = 1...3$ and represents the ISN, IPID, and RCVWIN, respectively. Note that A_1 and B_1 are encoded with their respective 16-bit random salts.

The internal structure of each 16-bit integrity field A_n and B_n is shown below the timing diagram in Figure 1. Integrity values in the forward path from A contain a 12-bit hash "fragment" and a 4-bit coverage type (*cvr*). The coverage type communicates which portions of the packet header are to be tested, and the same value is copied to each A_n. Coverage types are detailed in §4.4.

Similarly, integrity values sent from B each contain a 12-bit hash fragment over packet header fields in the SYN-ACK, and 3 bits to return the forward path integrity results to A. A examines these status bits in the received SYN-ACK to infer how its SYN arrived at B. To minimally impact the initial flow control window, the highest order bit of B_3 can be set to correspond to the true receive window. HICCUPS does not overload the window size field outside of the 3WHS.

In this paper, we abstract the integrity functions used to compute each 12-bit hash fragment as $f_n(\cdot)$. Thus $f_n(SYN, cvr)$ is the n'th integrity over the *cvr* fields in the SYN packet. The integrity function must be public, allowing the host at the other end of the connection, B, to check the integrity value it receives. Our experimentally validated [16] implementation in Linux uses a combination of truncated CRC32 and Murmur3 [4]. However, HICCUPS could be standardized to use different functions in the future, based on diffusion and collision-resistance requirements.

Table 2 lists possible inferences A and B can make during connection establishment. When B receives the SYN from A, it recomputes each A'_n using the SYN header fields as re-

Table 2: Possible knowledge gained by each host performing the integrity check

At B after receiving SYN	Inference
$\lvert A'_n = A_n \rvert \geq 2\ \forall n$	covered SYN fields unmodified
else	SYN modified or A not capable

At A after SYN-ACK recv'd	Inference
$\lvert B'_n = B_n \rvert \geq 2\ \forall n$	SYN-ACK unmodified
$\sum status_i \geq 2\ \forall status \in B_n$	SYN unmodified
Both cases above	SYN & SYN-ACK unmodified
else	SYN & SYN-ACK modified; or B not capable

ceived for each of the specified coverage types. The received integrity A'_n matches the sent integrity if $A'_n = A_n$. If at least two of the three recalculated hashes match the received hashes, B infers that the covered fields in A's packet header were unmodified in transit.

Next, B generates its own (different) salt and integrity values for the return SYN-ACK packet. B's results from verifying each A'_n are echoed back to A by the inclusion of boolean flags for each of ISN, IPID, and RCVWIN in the SYN-ACK integrity B_n. When A receives the SYN-ACK reply from B, it can also check the integrity values. A examines the forward path status bits to determine whether the SYN experienced manipulations.

Using $n = 3$ integrity fields and a combination of hash functions is crucial given the size limits (12 bits each). HICCUPS infers a packet as HICCUPS-capable when any two integrity values match the locally computed integrity ($A'_n = A_n$). Thus, the probability of a pre-image other than the original generating the same hash with two different hash functions is 2^{-24}, or approximately one in 16M. While this rate is non-negligible, it is low enough for practical use. Measurement instances requiring higher precision can run a HICCUPS integrity test multiple times.

4.4 What Header Field Was Modified

HICCUPS allows the connection initiator to specify which packet header field or subset of fields the handshake should check. For instance, a HICCUPS-enabled host opening a new connection could choose to only check the TCP MSS option, or it could focus on just the ECN flags. Each individual connection enabled with HICCUPS specifies which fields to check from a pre-defined list. HICCUPS currently supports the 16 coverage types shown in Table 3. A type that covers both the IP and TCP options blocks can be used to check other options. Our primary reasoning behind these design choices is directed by the highly constrained amount of space (we require the upper bits of B_n for forward path status) and the initiator being the party that typically chooses which options to negotiate for the connection.

All header fields, except for those that are expected to change in transit (e.g., TTL) or fields used to carry integrity, can be covered by HICCUPS. These immutable fields are denoted with a solid gray background in Figure 2. The `HFULL` type is the broadest and covers all of the immutable fields. The remainder of the coverage types we have implemented are proper subsets of these fields.

In order to check multiple types, a progression of HICCUPS connections can be performed between two endpoints. In this progression, each individual connection uses one of

Table 3: Pre-defined coverage sets

	Coverage Type	Header fields that are covered
0	`HNONAT`	Everything, minus IPs and ports
1	`HFULL`	Everything
2	`HNAT`	IPs and ports
3	`HNOOPT`	`HNONAT` minus any IP or TCP options
4	`HONLYOPT`	IP and TCP options
5	`HECNIP`	ECN IP codepoint
6	`HECNTCP`	ECE and CWR TCP flags
7	`HLEN`	Length fields
8	`HMSS`	TCP MSS option
9	`HWINSCL`	TCP Window Scaling option
10	`HTSTAMP`	TCP Timestamp option
11	`HMPTCP`	TCP Multipath option
12	`HEXOPT`	An unused TCP option (kind = 99)
13	`HFLAGS`	IP_DF, non-ECN TCP flags, and TCP SACK_Permitted option
14	`HSAFE`	Reserved fields, protocol, and version
15	`HNULL`	Nothing (compatibility check)

the pre-defined coverage sets. The simplest approach is to check all possible coverages in order. Such an approach would require a separate connection for each, but could be done in parallel to reduce the latency of multiple RTTs waiting for results. Alternatively, the inferences might occur during the natural interaction and multiple connections between hosts. A smarter algorithm that could reduce the total number of connections required is described in §5.5.

Selection of a coverage type for a given connection can be done manually by an application (§4.6) or automatically by the TCP stack. Once a type has been selected, we concatenate the covered packet header fields as input to the HICCUPS integrity functions $f_n(\cdot)$. The only exception is the two bits in the IP header that represent an ECN codepoint. For these two bits, we include their bitwise `OR` as input. Routers are allowed to modify this field, but only by turning an ECT0,1 codepoint into a CE codepoint. Nothing should set both bits to zero if either one was originally set high by an endpoint (an aberration observed in [6]).

Because a field carrying the integrity, A_n, could be modified, the endpoint analyzing the SYN must test all the coverage types it sees in the received A'_n. Ideally, none of A_n will have been overwritten meaning all three coverage values are the same and only one check must be done. The worst case is that three checks must be done in the event that one or more of A_n were overwritten. If the receiving endpoint finds a match, it must use the same coverage type when calculating B_n for the SYN/ACK. Should the receiver fail to find a match (meaning part of the SYN was modified), a majority rule is used on the three coverage types listed in A'_n to determine the coverage to use for B_n. If a majority is not found, a special coverage type is used in B_n to indicate to host A that at least two of A_n were modified.

4.5 AppSalt Protection

HICCUPS is designed to be cooperative with middleboxes. Unlike with checksums, packets will not be rejected by a host due to incorrect HICCUPS integrity. Our primary goal is to allow TCP endpoints to choose their extensions based on whether the path will support their correct interpretation end-to-end. By not providing middleboxes with a reason to disrupt HICCUPS, overwriting and recomputation of the integrity fields by middleboxes should be uncommon.

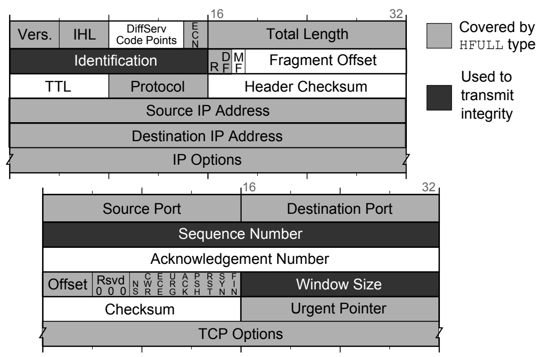

Figure 2: Header coverage by the `HFULL` probe

However, we recognize that future middleboxes, armed with knowledge of HICCUPS, may attempt to recompute hashes in an effort to induce endpoints into a false inference of path integrity. As a result, we designed HICCUPS with an optional, enhanced mode that we term "AppSalt."

AppSalt aims to make undetectable packet header manipulation expensive for a middlebox. With AppSalt, a middlebox must either (*i*) bear the cost of circumvention, (*ii*) reveal the modifications it makes to the endpoints or (*iii*) simply stop meddling in the communication. The value proposition of such a protocol is that (*i*) presents a high enough cost that the middlebox naturally chooses approach (*ii*) or (*iii*).

A middlebox, M, could disguise a packet header modification by rewriting the integrity values on SYNs from host A. Should M also want to modify the SYN-ACK response, it would perform its changes and then recalculate new integrity for the SYN-ACK sent by B. This situation could lead to the reduced effectiveness of HICCUPS at detecting potential extension compatibility issues as middleboxes adjust to evade HICCUPS, but then either fail to properly support newer extensions or suffer from a future misconfiguration.

Since our design constraints preclude the use of a stronger construction, e.g., a keyed-HMAC, we cannot outright prevent M from splitting the connection and recalculating valid integrity values for arbitrary packet header manipulations.

Instead, in AppSalt mode, HICCUPS protects integrity values by encoding them with a property of the connection that is only revealed *after* the 3WHS is complete. Such an "ephemeral secret" could be any property of a connection known only to the sender at the start of the connection.

From the perspective of the middlebox and receiver, the encoded integrity values in the three HICCUPS fields remain indistinguishable from random numbers until the ephemeral secret is revealed later in the connection. Thus, we are able to force a middlebox seeking to recompute our hashes to commit to a strategy *before* it even knows if the connection is HICCUPS-enabled. Since a HICCUPS-enabled TCP need not necessarily perform HICCUPS with every connection request, it is difficult for a middlebox to know when it should try to recompute new hashes. We thus add protection to the integrity while imposing as little of the increased burden as possible on the endhosts. The sending host only has to encode the integrity value and the receiving host only has to store the received integrity until the secret is revealed.

Both the future timing of packets and the number of packets in a flow are possible ephemeral secrets, yet those are difficult to control. Our HICCUPS implementation protects the SYN integrity values with future *application-layer content* from a data packet *yet to be sent*, an ephemeral secret

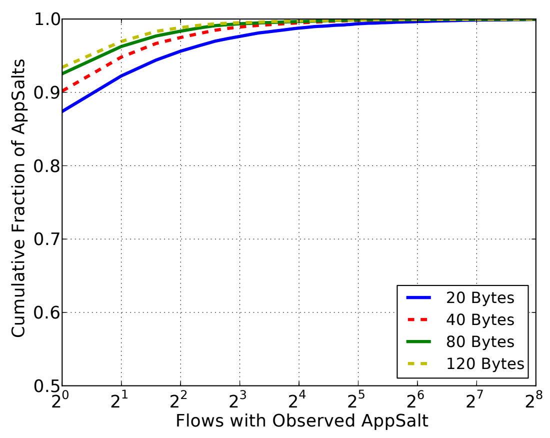

Figure 3: Cumulative fraction of application-layer payloads ("AppSalts") of different lengths versus number of flows in which the AppSalt appears.

that is difficult for a middlebox to reliably determine *a priori*. As in §4.3, the integrity values are placed in the ISN, IPID, and RCVWIN of the SYN, but now the receiving endhost, as well as any middleboxes, must know the contents of future application data in order to interpret the integrity.

For the ephemeral application-layer secret, we use a small portion of the data contained in the first data packet to make it simple for the receiver to locate and extract the AppSalt secret. We therefore examined the initial application payload of each flow in a full day of border traffic from our organization. Among application data payloads of 6,742,466 flows, we find 5,377,440 ($\approx$ 80%) where the first 40 bytes are unique. The 99th percentile of the distribution is that payloads appear twice, implying that 40 bytes of ephemeral secret is a reasonable lower-bound to prevent trivial guessing. Figure 3 shows the distributions for various lengths across a 30 minute capture.

To illustrate AppSalt operation, we present a scenario where a client connects to a webserver by performing the 3WHS and issues an HTTP GET request for a specific resource. Neither the remote server nor any in-path middleboxes can reliably determine the application data at the time the SYN is observed. Only the client knows with certainty the initial HTTP application data that will be sent. In this example, the application layer data might contain such items as the GET URL, the host parameter, and the user agent string as shown in the example of Figure 4.

Since the application data needed to properly decode the SYN's integrity is not available to M at the time the SYN is received, it is difficult for M to make an undetectable header modification or even just to check whether the connection is HICCUPS-enabled. The ephemeral secret forces M to process the SYN packet before it can observe the application data. Otherwise, M has two remaining options if its goal is to modify the packet headers and evade detection: make a best guess of the application data, or perform a man-in-the-middle (MITM) attack and fake a SYN-ACK response, inducing A to expose the application data secret.

M may attempt to guess the unseen application data, e.g., by using a profile of prior connections from A to B. However, M is unlikely to guess correctly for every connection between all pairs of hosts. If M guesses incorrectly, integrity values will not validate and the manipulations can be detected. Of course, M could later change the actual application data

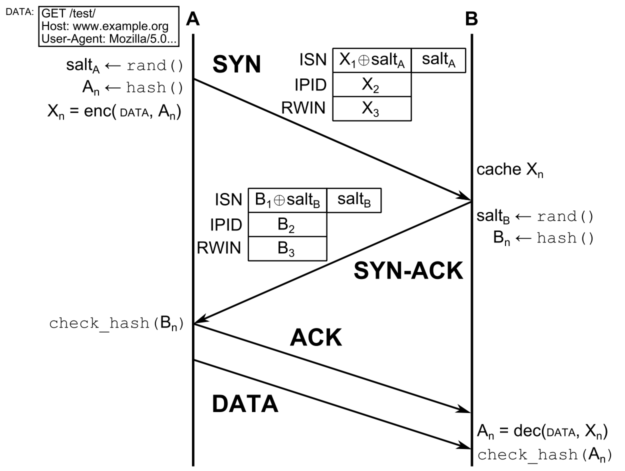

Figure 4: HICCUPS AppSalt protection: the integrity values in the SYN are encoded with application-layer data *yet to be sent*, forming an ephemeral secret that raises the bar on middleboxes attempting to evade HICCUPS diagnostics.

to match its guess, but doing so fundamentally alters the application-layer behavior of the connection.

In order to know the application data with certainty, M must act as a TCP-terminating proxy, a behavior that is detectable based on timing and by issuing connections to known unreachable hosts as shown in [31]. This MITM behavior, whereby M falsely claims to be B, spoofs the SYN-ACK and intercepts the resulting traffic, permits M to rebuild the original SYN with an updated integrity value and forward it along to the true destination. The non-spoofed SYN-ACK from B must be intercepted and the cached data from A could be sent. This situation is more complicated than just rebuilding the integrity values; the middlebox has broken a connection and now has to marshal data between them, in addition to sending spoofed packets and buffering data. Further, the middlebox must do this for all connections, potentially representing many endpoints.

AppSalt represents our proactive approach to ensuring the continued effectiveness of HICCUPS once its algorithms and protocol become widely known. Another possible disruption technique is to perform a downgrade attack by arbitrarily overwriting all fields used by HICCUPS for integrity. This does not circumvent the tamper-evidence, however, and the downgrade fails when there is outside *a priori* knowledge that the remote end is performing HICCUPS.

4.6 API

We have implemented HICCUPS as a patch to Linux kernel 3.9 [15]. We allow applications to request a certain coverage via a `setsockopt()` call specifying their desired coverage type (§4.4). Similarly, applications can read results of a HICCUPS diagnostic from the kernel with `getsockopt()`.

The use of AppSalt mode requires a minor change to the sockets API. Traditionally, a client program issues a series of socket calls: `socket()`, `connect()`, and `send()`. However, with AppSalt, `connect()` cannot be called first as it will initiate the 3WHS and send the SYN before the kernel has the necessary application data over which to calculate integrity.

We therefore leverage the same socket API changes implemented by TCP Fast Open (TFO), a TCP modification that similarly requires data be known at the time of connection initiation [39]. Programs that use TFO initiate all connections using `sendto()` or `sendmsg()` with the `MSG_FASTOPEN` flag, as opposed to the typical `connect()` and `send()` sequence. In this way, the kernel can embed data in the SYN for connections with a valid TFO cookie.

To allow a client program to request AppSalt-mode HICCUPS, we add a new message flag within the framework established by TFO: "`MSG_HICCUPS`." This implementation style makes the addition of HICCUPS support trivial for applications that already support TFO, e.g., Google Chrome [23]. If application data cannot be used, i.e., a program does not use the new socket calls or it is a TFO connection with data in the SYN, plain HICCUPS is used instead (as in Figure 1).

5. RESULTS

This section details results from running HICCUPS in the wild. We examine the types, frequencies, and symmetry of HICCUPS-inferred modifications and give examples of how a TCP HICCUPS instance can adjust its behavior based on path inference to improve performance. Last, we discuss HICCUPS overhead, including the empirical number of RTTs for full-path characterization.

5.1 Controlled Environment

To test the validity of HICCUPS inferences, we validated against known ground truth in a controlled laboratory environment. Using NFQUEUE [10] and Scapy [7], we simulated a middlebox that makes a variety of packet header modifications [16]. On virtual machines running the HICCUPS kernel we performed 50,000 trials that established 3.2 million TCP connections—all traversing the middlebox simulator. Automated verification found that HICCUPS properly inferred the path behavior for 100% of the connections.

5.2 Overhead

We examined server-side overhead associated with HICCUPS using the Linux kernel's ftrace facility. Taking the average over 1000 connection attempts, we compared the total time spent processing a SYN/ACK between the HICCUPS-patched kernel and a vanilla kernel. We found that the average overhead added by our unoptimized implementation is about 8.5% of the compute time in the vanilla kernel.

Should a server begin to exhaust its resources (possibly due to a SYN flood or denial-of-service attack), mitigation methods are already available in the kernel to reduce this overhead. As the connection backlog fills, Linux can switch from processing HICCUPS checks on incoming SYNs to creating SYN cookies. While SYN cookies and HICCUPS cannot be used at the same time, they can still gracefully coexist since the situations where they perform best do not overlap.

5.3 Surveying Internet Paths with HICCUPS

While previous research (e.g., [6, 17, 25, 31, 36]) examined real Internet paths to catalog various forms of packet header modifications, these efforts required some degree of interaction external to the operating systems. To our knowledge, HICCUPS is the first solution to both capture measurements of packet header modifications within TCP and expose the results *directly through the operating system itself.* For example, the servers in our measurement infrastructure do not run any specialized server application. Instead, we simply start a standard HTTP daemon that listens on the desired

Table 4: Top ASNs represented

Servers		PlanetLab		Ark	
AS16509	6	AS680	13	AS22773	3
. . .	1 ea.	AS2200	6	AS1213	2
		AS766	6	. . .	1 ea.
		. . .	<6 ea.		
Total	7	Total	154	Total	53

Table 5: Geographic distribution

Location	PlanetLab	Ark	Servers
Europe	101	18	1
N. America	75	25	7
Asia	26	9	2
S. America	10	1	1
Oceania	6	0	1
Africa	0	3	0
Total	218	56	12

port(s). With a HICCUPS-enabled kernel, no extra support is required to perform HICCUPS and expose path behaviors to the operating system and applications.

Using HICCUPS-enabled hosts, we survey a diverse set of real Internet paths. We employ 218 Planetlab [11] nodes, 56 Archipelago (Ark) [26] nodes, and 12 distributed HICCUPS servers; the Autonomous System (ASN) and geographic distribution of our infrastructure is given in Tables 4 and 5. This infrastructure enables us to run HICCUPS between 3,288 pairs of distinct hosts, testing 26,304 directed path/port pairs.

5.3.1 Experimental Infrastructure

Our HICCUPS-enabled Linux kernel runs on 12 systems: four at the authors' institutions and one at each of the eight Amazon EC2 infrastructure sites. To run HICCUPS from PlanetLab (where installing a custom Linux kernel is not possible), we duplicate the connection initiation portion of TCP with HICCUPS into a user-space client that employs raw sockets to craft HICCUPS-enabled SYNs.

In selecting PlanetLab nodes, we used PlanetLab's management API to use a single node per site. Thus, all 218 Planetlab nodes we use represent distinct sites. The PlanetLab nodes were distributed both geographically and logically around the Internet. The Planetlab and Ark nodes reside in 207 distinct ASNs. Geographically, our Planetlab nodes are situated in five continents and 37 different countries, while the Ark nodes are spread across 28 countries.

5.3.2 Experimental Parameters

From each PlanetLab and Ark vantage point, we execute SYN exchanges with each server on four different TCP ports to capture port-specific behavior: 22, 80, 443, and 34343. The first three are common service ports; port 34343 is used for consistency with [25]. We send 16 SYNs to each of the four ports, with each SYN covering one of the different coverage types listed in Table 3. Note that not all paths require all 16 connections to fully ascertain the path conditions from HICCUPS; ~90% of paths can be fully characterized in two RTTs. We examine this aspect further in §5.5.

To make middlebox modification behaviors visible, we must enable different TCP and IP extensions during the connection setup. Table 6 lists the sets of options we use in our experiments, including MSS, SACK permitted, Window Scale, Timestamp [27], Multipath TCP MPCAPABLE [20], and a non-standard experimental option with a kind value of `99`.

Table 6: Experimental parameters for each trial

Trial	MSS	ECN	SACK Permit	Win Scale	Time stamp	MP-TCP	Exp
1	1460		Y	7	Y		Y
2	1460		Y	7	Y	Y	
3	1460		Y	7	Y		
4	1460	Y					
5	480						
6	1460						
7	1600						
8	None						

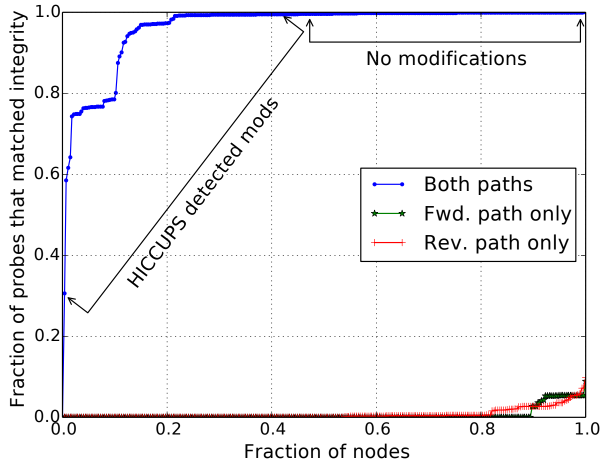

Figure 5: Distribution of matching probes, by path direction. NAT modifications have been excluded.

5.4 Detected Modifications

Following the inference procedure in §4.3 and Table 2, we use HICCUPS to detect a variety of packet header manipulations. If a probe passes integrity checks at the receiver and the forward path status bits return intact, the TCP initiator infers that its packets (on the forward path) arrive without modification. Similarly, if the integrity checks on the SYN-ACK match, the initiator infers that the reverse path does not modify headers. All data we present in this section comes from the clients on PlanetLab and Ark.

Figure 5 displays the cumulative fraction of probes per host with passing integrity versus the fraction of nodes (PlanetLab and Ark nodes combined with NAT results excluded). The common case is that both the forward and reverse paths experience no modifications. For approximately half of the nodes, all probes match integrity, while approximately 80% of the nodes have 99% of their probes match integrity. The distributions for the two asymmetrical integrity results are visible in the lower-right of the figure. Approximately 80% of nodes never experience this case.

Table 7 summarizes the probe results according to coverage type. The most common modification is paths that add or change MSS values. The HNAT—and consequently the inclusive HFULL probe—fails for the large majority of paths. This is unsurprising as address translation is performed near the server for 9 of our 12 servers. We verified that while our Amazon EC2 servers experienced NAT, they made no other header modifications. In the following subsections, we more closely examine specific modifications.

5.4.1 ISN translation

We find incidences of sequence number translation in tests from 24 of 218 Planetlab nodes (11.0%). The ISNs are trans-

Table 7: Summary of results by coverage type

Coverage	Integrity Match				
	Both	Fwd	Rev	Neither	Timeout
HFULL	21867	597	985	80931	836
HNAT	25286	2	0	79129	799
HNONAT	91214	2397	2459	8329	817
HNOOPT	100535	71	2050	1732	828
HONLYOPT	92948	2542	1162	7736	828
HECNIP	102066	69	1693	572	816
HECNTCP	103777	10	47	585	797
HLEN	103451	17	359	574	815
HMSS	93365	2545	855	7632	819
HWINSCL	103685	16	5	690	820
HTSTAMP	103834	27	7	539	809
HMPTCP	103023	20	837	551	785
HEXOPT	102907	12	888	564	845
HFLAGS	102591	18	76	1719	812
HSAFE	103824	16	0	551	825
HNULL	103752	21	0	563	880
Total	1458125	8380	11423	192397	13131

lated in both directions on 20 nodes, while for four nodes, just the forward path translates sequence numbers. Only one of the Ark nodes is subject to ISN translation that occurs on forward path only.

The frequent occurrence of sequence number translation motivates in part our choice to use three hash fragments, as detailed in §4.3. If, for instance, the ISN alone carried integrity, HICCUPS would not work for 25 of our 274 nodes and we would be unable to detect any header modifications beyond ISN translation. In contrast, HICCUPS can withstand a single modification to any one of the three integrity-carrying fields (ISN, IPID, and RCVWIN).

However, should any pair of the three fields be modified, HICCUPS loses the capability to detect specific field modifications, only noting that a change occurred to at least one pair of the three integrity fields. Table 8 lists paths where this behavior occurs under the heading “HICCUPS not capable.” 68 flows from PlanetLab (0.7%) and 4 flows from Ark (0.2%) saw two or more integrity fields overwritten. Since we control all the nodes, we performed post-mortem analysis of packet captures taken during measurement and see that the TCP receive window is artificially lowered in-path. In practical use, however, HICCUPS cannot obtain any fine-grained information for such paths.

5.4.2 ECN

We monitor behavior of the ECN fields in both the IP and TCP headers. Figure 6 shows the results of each probe arranged by host in the combined PlanetLab and Ark datasets. Each of the three plots in the figure represents the results from probing each of the 48 server ports from each of the 274 nodes. Each plot is sorted so that primary result types are grouped together. The first plot shows the behavior when ECN was disabled, while the lower two show behavior after ECN has been enabled. While ECE and CWR TCP flags are rarely affected (we only saw such mods on paths from one PlanetLab node), modifications to the IP codepoint are more common. We observed ∼13% of paths on both PlanetLab and Ark would zero the codepoint if it were enabled.

5.4.3 Application Performance

An important consequence of HICCUPS is that knowledge of the end-to-end header modification state of a path can improve the performance of applications that depend on

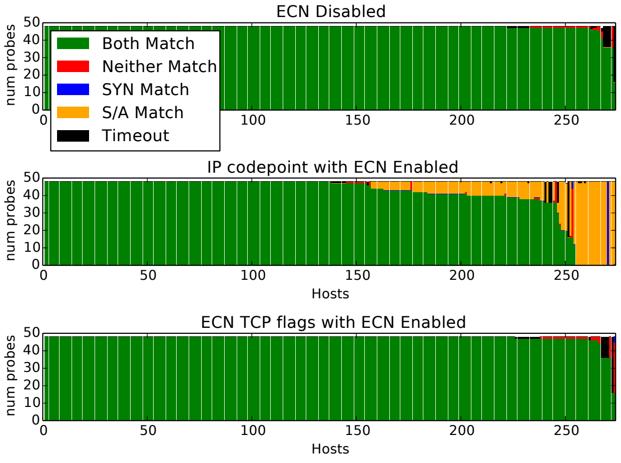

Figure 6: Distribution of HICCUPS-inferred ECN path properties. For the IP codepoint, HICCUPS only notes a change to the `OR` of the bits (§4.4).

TCP. For instance, in the case of sequence number translation that is SACK-naïve, performance suffers in proportion to loss rate [24]. For ECN, performance suffers when false congestion signals are inadvertently marked, experiencing dramatic performance impact if a congestion codepoint is added, or a TCP-layer congestion echo is added [6]. To highlight the potential impact on TCP performance, we examine a particular effect, observed in the wild, in detail.

We find a node where the forward communication transparently adds a TCP window scale value of 7 to the SYN, but the reverse path strips the window scale by replacing it with 4 `NOP` options in the returned SYN-ACK. The behavior is destination port-specific: it did not occur on connection attempts to ports 22 or 34343, only to 80 and 443. Ultimately, one end of the communication believes that window scaling negotiation has occurred, while the other does not.

We perform bulk transfer to the node performing window scaling and observe that the traffic is flow controlled—the receiver is sending scaled values in the receive window, but the sender interprets those values as unscaled. HICCUPS informs us of the option mangling and we disable window scaling. Our performance tests reveal a dramatic difference where the throughput more than doubles without window scaling since the congestion window can open more than one or two MSS. We alerted the operator of the node and they were unaware of the behavior. Further investigation revealed the issue was with a system in their provider’s network.

5.5 Complete Path Knowledge

Given that only one coverage set from §4.4 is used per TCP 3WHS, a pair of TCP endpoints must develop fully granular knowledge of all header modifications over the course of multiple exchanges. When integrity matches for a coverage type that is a superset of other types, e.g., `HFULL`, no further information is gained from additional probing. However, if the integrity fails to match, more specific types can be used next to narrow down the source of the modification.

If integrity using `HNULL` does not match, then either one of two cases is occurring: (i) two or more of our three integrity fields are being modified, or (ii) the host with which we are interacting does not understand HICCUPS. Since `HNULL` is a diagnostic type that does not cover other fields, it should not fail unless the hash fragments are not present.

Table 8: Summary of HICCUPS-inferred header modifications. Detection of ISN, IPID, and RCVWIN are mutually exclusive to HICCUPS. If two or three occurred, it registered as "HICCUPS not capable" instead.

	Planetlab					Ark				
Change	**Both**	**Fwd**	**Rev**	**Flows**	**Affected**	**Both**	**Fwd**	**Rev**	**Flows**	**Affected**
HICCUPS not capable	68	0	2	10360	0.68%	4	0	0	2684	0.15%
NAT	7704	0	0	10281	74.93%	2114	0	0	2677	78.97%
ISN translation	924	178	0	10290	10.71%	0	48	0	2680	1.79%
IPID change	0	0	0	10290	0.00%	0	0	0	2680	0.00%
RCVWIN change	0	0	0	10290	0.00%	0	0	0	2680	0.00%
ECN IP add	26	0	0	10270	0.25%	2	0	0	2664	0.08%
ECN IP change	16	1342	48	10283	13.67%	11	342	0	2675	13.20%
ECN TCP add	16	0	0	10261	0.16%	6	0	0	2670	0.22%
ECN TCP change	19	46	0	10285	0.63%	16	0	0	2675	0.60%
MSS add	119	47	1036	10258	11.72%	10	96	140	2668	9.22%
MSS480 change	21	0	1132	10281	11.21%	5	0	139	2674	5.39%
MSS1460 change	1113	0	0	10275	10.83%	134	12	12	2678	5.90%
MSS1600 change	1105	157	0	10294	12.26%	140	154	12	2672	11.45%
SACK Permit changed	1	24	0	10123	0.25%	0	0	0	2667	0.00%
Timestamps add	12	0	0	10267	0.12%	9	0	0	2669	0.34%
Timestamps change	26	2	0	10279	0.27%	10	0	0	2672	0.37%
Window Scaling add	45	0	0	10265	0.44%	9	0	0	2665	0.34%
Window Scaling change	24	0	0	10279	0.23%	5	0	0	2669	0.19%
`MPCAPABLE` change	24	837	0	10267	8.39%	8	0	0	2673	0.30%
Exp. option change	20	884	0	10266	8.81%	13	0	0	2676	0.49%

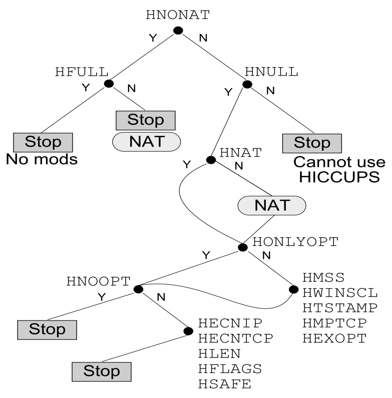

Figure 7: HICCUPS Search Strategy

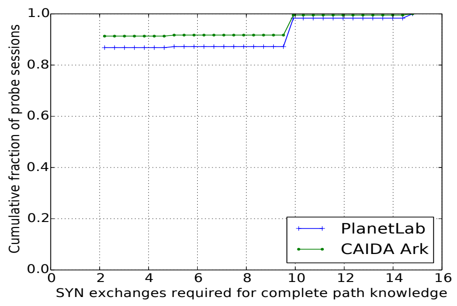

Figure 8: Empirical HICCUPS RTTs required for complete path properties inference

Leveraging this information, we design a path interrogation strategy for HICCUPS. Using HICCUPS to determine the fully granular set of modifications along a path is similar in nature to a search problem. Our informed strategy is shown in Figure 7. We begin by checking coverages that are more comprehensive and then narrow the search, eventually checking a smaller sequence of types. Upon our first interaction with a TCP endpoint, we choose the `HNONAT` coverage type since it avoids fields modified by NATs, which are prevalent on the Internet [31]. If we find a match, we conclude the search. Subsequent connection attempts can retest using the `HNONAT` type in case the path conditions change.

Given that we expect regular interaction with non-HICCUPS TCP stacks, our strategy employs the `HNULL` type at the next opportunity. By doing so, we can terminate the search in the event that either the other endpoint (due to lack of capability) or middleboxes along the path (due to downgrading the integrity) prevent HICCUPS from being used. The remainder of the strategy searches for header modifications in either the options space or fixed-length fields, iterating through a series of more granular coverage types as needed.

5.5.1 *Expected Interactions Required*

Across real paths in our PlanetLab and Ark datasets, we calculated the number of TCP interactions it would take for two HICCUPS hosts to fully ascertain the path header modification state. For PlanetLab, our dataset contained 83,712 flows with 261,185 total SYN exchanges required to fully explore the space of header modifications with HICCUPS. This amounts to an average of 3.1 SYN exchanges per flow. For Ark, we required 58,083 SYN exchanges across a total of 21,504 flows, for an average of 2.7 exchanges per flow.

Figure 8 shows that about 85% of flows were able to fully determine the modifications of their paths after checking just `HNONAT` and `HFULL`. Should NAT detection not be desired, the check for `HFULL` could be omitted from the strategy shown in Figure 7, further reducing the required number of probes.

6. CONCLUSIONS

Debugging IP network problems end-to-end is a difficult, often manual process exacerbated by the presence of currently opaque middleboxes. We present TCP HICCUPS, a backward-compatible and incrementally deployable extension to TCP that reveals packet header manipulation to both sides of a TCP connection, enabling endpoints to make the inferences needed to best adapt to middleboxes along their paths. For example, we show how HICCUPS helps achieve twice the throughput over a TCP naïve to paths that modify window scaling. HICCUPS can also help facilitate the safe deployment of new and experimental options.

Beyond improving TCP performance, widespread HICCUPS deployment could provide invaluable data to researchers, policy makers, and protocol designers. Measurements from running HICCUPS across a distributed and diverse set of paths discover a wide variety of (sometimes asymmetric) behaviors, including paths that modify, delete, or insert: sequence number, IPID or receive window, ECN, MSS, timestamps, window scaling, Multipath TCP, and an experimental option. Crucially, header modification behaviors are discovered by a HICCUPS-enabled TCP stack without prior coordination from the remote endpoint. Such a usage model also enables new diagnostic capabilities for network operators to help troubleshoot middlebox configurations on both forward and reverse data planes.

In future work, we wish to refine the efficient search strategy used by HICCUPS to granulate header modifications by field. We plan integration with response algorithms for TCP to automate the performance gains that HICCUPS inferences enable. To this end, we plan a more extensive performance characterization of selectively toggling extensions in response to behavior inferred by HICCUPS. Another approach we will pursue is to examine how some middleboxes, such as the array of proxy devices in mobile networks, could utilize and safely interact with HICCUPS integrity information. Last, we wish to continue our survey of Internet paths, analyzing header modifications and their impact over many more types of paths and investigating the potential to characterize middleboxes by the modifications they induce, e.g., TCP `NOP` options that are not required for alignment.

Acknowledgments

We thank Geoff Xie, Nick Weaver, Mark Gondree, Justin Rohrer, and our shepherd Vivek Pai. Steve Bauer, Young Hyun, and Mark Richer provided infrastructure and testing. This work is supported in part by National Science Foundation (NSF) grants CNS-1213155, CNS-1213157, and CNS-1237265, and SPAWAR Systems Center Atlantic NISE. This material represents the position of the authors and does not reflect the official policy or position of the U.S. Government.

7. REFERENCES

[1] ABI. Enterprise network and data security spending shows remarkable resilience, Jan. 2011. http://goo.gl/E5Unmb.

[2] M. Allman. Comments on Selecting Ephemeral Ports. *SIGCOMM Comput. Commun. Rev.*, 39(2):13–19, Mar. 2009.

[3] Anonymous. Private communication, 2011.

[4] A. Appleby. MurmurHash 3.0, 2011.

[5] F. Baker. Requirements for IPv4 routers. RFC 1812, 1995.

[6] S. Bauer, R. Beverly, and A. Berger. Measuring the State of ECN Readiness in Servers, Clients, and Routers. In *Proceedings of the ACM SIGCOMM IMC*, pages 171–180, Nov. 2011.

[7] P. Biondi. Scapy. http://goo.gl/aTHPX8.

[8] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. The case for ubiquitous transport-level encryption. In *Proc. of the USENIX Security Symposium*, Aug. 2010.

[9] B. Carpenter and S. Brim. Middleboxes: Taxonomy and issues. RFC 3234, Feb. 2002.

[10] P. Chifflier. nfqueue-bindings. http://goo.gl/OOmFi9.

[11] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, July 2003.

[12] Cisco Systems. Single TCP flow performance on firewall services module (FWSM), Oct. 2011. http://goo.gl/GktT8Z.

[13] D. Clark. The design philosophy of the DARPA internet protocols. *SIGCOMM CCR*, 18(4):106–114, Aug. 1988.

[14] E. Cole. *Hiding in Plain Sight: Steganography and the Art of Covert Communication*. Wiley Publishing Inc., 2003.

[15] R. Craven, R. Beverly, and M. Allman. Handshake-based Integrity Check of Critical Underlying Protocol Semantics (HICCUPS), 2014. http://tcphiccups.org.

[16] R. Craven, R. Beverly, and M. Allman. Techniques for the detection of faulty packet header modifications. Technical Report NPS-CS-14-002, Naval Postgraduate School, Mar. 2014.

[17] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing Middlebox Interference with Tracebox. In *Proc. of the ACM SIGCOMM IMC*, pages 1–8, Oct. 2013.

[18] M. Dischinger, M. Marcon, S. Guha, P. K. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling End Users to Detect Traffic Differentiation. In *USENIX NSDI*, 2010.

[19] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. IP Options are not an option. Technical Report 2005-24, EECS UC Berkeley, Dec. 2005.

[20] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP extensions for multipath operation with multiple addresses. RFC 6824, Jan. 2013.

[21] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, Aug. 2011.

[22] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward Software-Defined Middlebox Networking. In *Proc. of the ACM HotNets Workshop*, Oct. 2012.

[23] Google Inc. chromium code search, 2013. http://goo.gl/8PQrpG.

[24] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure. Are TCP Extensions Middlebox-proof? In *Proc. of the HotMiddlebox Workshop*, pages 37–42, 2013.

[25] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it Still Possible to Extend TCP? In *Proc. of the ACM SIGCOMM IMC*, pages 181–194, 2011.

[26] Y. Hyun and k. claffy. Archipelago (Ark) measurement infrastructure. CAIDA, 2014. http://goo.gl/HY9AgZ.

[27] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, May 1992.

[28] S. Kent. IP authentication header. RFC 4302, Dec. 2005.

[29] S. Kent and K. Seo. Security architecture for the Internet Protocol. RFC 4301, Dec. 2005.

[30] A. Knutsen, A. Ramaiah, and A. Ramasamy. TCP option for transparent Middlebox negotiation. Internet draft, Feb. 2013.

[31] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating The Edge Network. In *SIGCOMM IMC*, 2010.

[32] A. Langley. Opportunistic Encryption Everywhere. *Web 2.0 Security and Privacy (W2SP)*, May 2009.

[33] M. Luckie and B. Stasiewicz. Measuring Path MTU Discovery Behaviour. In *Proc. of the ACM SIGCOMM IMC*, 2010.

[34] D. Malone and M. Luckie. Analysis of ICMP Quotations. In *Proc. of PAM Conference*. Apr. 2007.

[35] M. Mathis and J. Heffner. Packetization layer path MTU discovery. RFC 4821, Mar. 2007.

[36] A. Medina, M. Allman, and S. Floyd. Measuring the Evolution of Transport Protocols in the Internet. *SIGCOMM Comput. Commun. Rev.*, 35(2):37–52, Apr. 2005.

[37] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers. RFC 2474, Dec. 1998.

[38] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. of the ACM SIGCOMM Conference*, Aug. 2013.

[39] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proc. of CoNEXT*, 2011.

[40] C. Reis, S. Gribble, T. Kohno, and N. Weaver. Detecting In-Flight Page Changes with Web Tripwires. In *Proc. of the USENIX Symposium on NSDI*, Apr. 2008.

[41] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *Proc. of the ACM HotNets Workshop*, 2011.

[42] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. of the ACM SIGCOMM Conference*, pages 13–24, Aug. 2012.

[43] J. Stone and C. Partridge. When the CRC and TCP checksum disagree. *SIGCOMM CCR*, 30(4):309–319, 2000.

[44] J. Touch, A. Mankin, and R. Bonica. The TCP authentication option. RFC 5925, June 2010.

[45] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful. In *Proc. of the USENIX Symposium on OSDI*, Dec. 2004.

[46] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. In *Proc. of the ACM SIGCOMM Conference*, pages 374–385, Aug. 2011.