Techniques for the detection of faulty packet header modifications

Craven, Ryan; Beverly, Robert; Allman, Mark

Monterey, California. Naval Postgraduate School

NPS-CS-14-002

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

TECHNIQUES FOR THE DETECTION OF FAULTY PACKET HEADER MODIFICATIONS

by

Ryan Craven
Robert Beverly
Mark Allman

March 12, 2014

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
12–03–2014	Technical Report	2013-05-01—2014-01-31

4. TITLE AND SUBTITLE

TECHNIQUES FOR THE DETECTION OF FAULTY PACKET HEADER MODIFICATIONS

5a. CONTRACT NUMBER

5b. GRANT NUMBER

CNS-1213155

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Ryan Craven, Robert Beverly, Mark Allman

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

NPS-CS-14-002

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Science Foundation
Arlington, VA 22230

SPAWAR Systems Center Atlantic
North Charleston, SC 29419

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT

Understanding, measuring, and debugging IP networks, particularly across administrative domains, is challenging.Compounding the problem are transparent in-path appliances and middleboxes that can be difficult to manage and sometimes left out-of-date or misconfigured.As a result, packet headers can be modified in unexpected ways, negatively impacting end-to-end performance.We discuss the impact of such packet header modifications, present an array of techniques for their detection, and define strategies to add tamper-evident protection to our detection techniques.We select a solution for implementation into the Linux TCP stack and use it to examine real-world Internet paths.We discover various instances of in-path modifications and extract lessons learned from them to help drive future design efforts.

15. SUBJECT TERMS

Middleboxes, network measurement, tamper-evident protocols

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Robert Beverly
Unclassified	Unclassified	Unclassified	UU	97	19b. TELEPHONE NUMBER *(include area code)* (831) 656-2132

NSN 7540-01-280-5500

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Ronald A. Route President	Douglas A. Hensler Provost

The report entitled "*Technique for the Detection of Faulty Packet Header Modifications*" was prepared for and funded by the National Science Foundation and SPAWAR Systems Center Atlantic.

Further distribution of all or part of this report is authorized.

This report was prepared by:

Ryan Craven

Robert Beverly

Mark Allman

Reviewed by:	**Released by:**
Peter J. Denning, Chairman Computer Science Department	Jeffrey D. Paduan Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Acronyms and Abbreviations

ACK acknowledgment

AH Authentication Header

BGP Border Gateway Protocol

CE Congestion Encountered

CRC cyclic redundancy check

CWR Congestion Window Reduced

DoS Denial of Service

DSCP Differentiated Services Code Point

ECE ECN Echo

ECN Explicit Congestion Notification

ECT ECN-Capable Transport

EFF Electronic Frontier Foundation

ICMP Internet Control Message Protocol

ICSI International Computer Science Institute

IDS intrusion detection system

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IP Internet Protocol

IPID IP identification

IPsec Internet Protocol Security

ISN	initial sequence number
ISP	Internet Service Provider
MAC	message authentication code
MSS	maximum segment size
NAT	network address translation
NS	nonce sum
PMTUD	Path MTU Discovery
RFC	Request for Comments
RTT	round trip time
SACK	selective acknowledgment
SDN	software-defined networking
SSL	Secure Sockets Layer
SYN	synchronize
TCP	Transmission Control Protocol
TCP-AO	TCP Authentication Option
TLS	Transport Layer Security
TOS	type of service
TTL	time-to-live
UDP	User Datagram Protocol

Abstract

Understanding, measuring, and debugging IP networks, particularly across administrative domains, is challenging. Compounding the problem are transparent in-path appliances and middleboxes that can be difficult to manage and sometimes left out-of-date or misconfigured. As a result, packet headers can be modified in unexpected ways, negatively impacting end-to-end performance. We discuss the impact of such packet header modifications, present an array of techniques for their detection, and define strategies to add tamper-evident protection to our detection techniques. We select a solution for implementation into the Linux TCP stack and use it to examine real-world Internet paths. We discover various instances of in-path modifications and extract lessons learned from them to help drive future design efforts.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Packet transit in the modern Internet is complicated by a diverse abundance of network devices that introduce modifications to a packet and its semantics. These modifications are possible because, aside from a few exceptions that will be discussed later, packet header control information is rarely encrypted or authenticated [1] and thus not protected as it traverses the network from one end host to another.

As we will show, packets may experience a variety of changes to their fields while in transit, both intentional and unintentional. Intentionally performed changes, such as a firewall rule desired by a network administrator, are an acknowledged fact of life on the Internet. Unintentional packet modifications, however, can be the result of misconfigurations or legacy devices and are an often under-appreciated issue that can have a widespread impact on the network. The primary motivation of this work is to expose these problematic changes to critical packet fields which can alter semantics and lead to unintended protocol interactions.

1.1 Middleboxes

Many different types of devices interact with a packet as it traverses the network. Traditionally, the primary objective of these devices was packet forwarding. It is increasingly common, however, for packets to encounter devices whose primary task is something other than just the forwarding or routing of packets. Request for Comments (RFC) 3234 defines a term for these types of devices, *middleboxes*, and gives a taxonomy of the various types that existed at the time of publishing [2].

1.1.1 Types of Middleboxes

Some examples of well-known and commonly deployed middleboxes include: firewalls, network address translation (NAT) devices, performance-enhancing proxies, and transcoders that modify image files to reduce their size. Systems such as these are very prevalent on the network, with each one having the ability to modify packets for its own purposes. Recent studies have shown that in networks of all sizes, the number of middleboxes is on par with the number of routers [3].

NAT devices, in particular, are extremely prevalent. Recent statistics from the network diagnostic tool Netalyzr show that about 90% of its sessions came from behind a NAT device [4]. They are nearly ubiquitous on home and corporate networks due to the decreasing availability of globally-routable IPv4 addresses.

Also in use on the Internet are a myriad of other more specialized “security-enhancing”

devices like sequence number randomizers [5], fingerprint scrubbers [6], active wardens [7, 8], and traffic normalizers [9]. Because these devices sit in the line, they can alter any bits within the entire packet header. NAT devices, for example, are expected to alter certain fields such as Internet Protocol (IP) addresses and TCP or User Datagram Protocol (UDP) port numbers, but nothing stops the same device from changing more than that, such as sequence numbers and IP or TCP options.

1.2 The Problem

Middleboxes are difficult to manage and maintain. Networks of all sizes employ a diverse set of middleboxes that serve a variety of purposes. Many are often from various vendors, usually run on separate physical hardware, and require configuration by a well-trained administrator. As a result, they can require a large support staff, which greatly adds to the already expensive cost of purchasing devices and their licenses [3].

All of these factors contribute to the introduction of misconfigured, nonstandard, or out-of-date legacy behaviors in middleboxes. In a survey of 57 network administrators, the majority overwhelmingly cited misconfiguration as the most common cause of middlebox failure, most likely due to the management and upgrade complexity involved [3].

With their prevalence and how much power they have to alter packets and their semantics, it is a problem when middleboxes operate incorrectly or make unintended changes. As we will show in Section 2.1, such changes can result in unexpected protocol interactions and end-to-end performance issues. Furthermore, even just the threat of encountering such issues can negatively influence protocol innovation, forcing designers to scale back improvements and take overly conservative deployment strategies [10–14]. As we show in Section 2.1 and confirm in Chapter 7, such end-to-end traversal issues still occur and are a real problem on the Internet.

The existing solution space for this problem can be broken down into three primary categories of approaches: prevention, avoidance, or detection. Prevention involves using strong cryptography to stop middleboxes from tampering with packet headers. Avoidance tries to fix the middleboxes themselves before issues arise. The final category, detection, includes solutions that check to see whether any modifications were made to the packet headers in-flight. Section 2.2 describes examples of related work that fall under each category and the limitations they have.

In this work, we advocate a strategy of detection. We feel that this strategy is the most flexible, most cooperative with current middleboxes, and will be the most likely type of approach to achieve wide adoption, which is the key to any solution.

1.3 Impact on Relevant Stakeholders

The significance of this work is the impact it could have on our understanding of the global Internet architecture. Successful adoption of this technology would not only make debugging and troubleshooting easier, but yield new insights about the integrity of packet transit across a large portion of the network. It would also enable interesting future measurement studies that could clarify the impact of middleboxes and allow network administrators to implement new protocols more safely, without alienating a small set of users behind a faulty middlebox. This could accelerate the deployment of new protocols that enhance robustness, reliability, or offer new functionality.

We also believe that many incentives exist for adoption of our proposed solution and could benefit all groups of key Internet stakeholders:

- End users and content providers want to know their traffic is treated fairly by their transit providers, i.e., network neutrality.
- Large Internet companies want to take advantage of as many protocol extensions as possible to increase performance, but are hesitant to enable them due to blackholes and middlebox interference.
- System and network administrators would have access to a new diagnostic tool to troubleshoot more complex connectivity issues.
- The networking community would be able to draw upon a wealth of new information for measurement studies to better understand the global Internet.
- Any end host enabled with our detection code could instantly become a cooperating end point in a path integrity test. This would be similar to how `ping` and `traceroute` are used today to enable network tests with a large number of hosts without requiring prior coordination with the system administrators.

1.4 Document Structure

The remainder of this report is organized as follows:

Chapter 2 discusses related research on middleboxes and the issues they can cause. We also survey the various existing solutions introduced in Section 1.2 and discuss their shortcomings.

Chapter 3 builds on the shortcomings we found with other solutions and defines the architectural design principles we feel must be achieved in an effective solution. These goals are later used to evaluate the design alternatives we propose.

Chapter 4 discusses methods for transmitting integrity information and the related design considerations. We also present the various designs we generated in the pursuit of finding the best solution.

Chapter 5 considers what can happen when a middlebox attempts to fake the integrity in-

formation and details various solutions to add protection to the integrity value.

Chapter 6 details our implementation in the Linux kernel.

Chapter 7 presents the results from experiments using our protocol on the Internet.

Chapter 8 concludes our work by summarizing key points and discussing opportunities for future work.

CHAPTER 2:
Background and Related Work

In this chapter, we begin by more closely examining the breadth of disruptions that can arise from misconfigured middleboxes, going into depth on several selected issues. We then survey the possible solution space, examining the current state-of-the-art in each of the three categories of approaches mentioned in Section 1.2. We find that each solution suffers from some key limitations that negate its efficacy for our problem domain. The shortcomings we find will be used later to build our architectural requirements in Chapter 3 to define where and how we can improve upon current solutions.

First, we look at network traffic integrity mechanisms in use today and discuss why they fall short in the context of our problem. We also discuss related work that approaches middlebox coordination from a different angle by attempting to redesign the middlebox architecture using elements of software-defined networking. Finally, we take a closer look at related detection-based solutions. The most viable alternative we found is able to detect in-path middlebox modifications, but is not integrated into TCP and has several limitations that we strive to overcome.

2.1 Middlebox Issues

Architectural issues with middleboxes and their unintended consequences are well-documented [2]. In 2004, Medina *et al.* detailed several issues caused by unexpected interactions on the part of a middlebox [15, 16]. Their tests show a mere 41% success rate of Path MTU Discovery (PMTUD), a network technique to avoid fragmentation by detecting the largest segment size allowed by a path. ICMP blocking by middleboxes was pinpointed as the presumable cause of failure for 18% of the servers tested. In addition, middleboxes were found to disrupt IP options; using a non-standard IP option resulted in failure of over 70% of connections, with mixed results even for common options. The concern of such behavior is not just that the options were not propagated, but even worse that the entire packet was dropped at some point along the path, i.e., the "blackhole" problem. The authors also found that Explicit Congestion Notification (ECN) was a problem area as well, where middleboxes overwrite flags, causing failures with negotiation and congestion notification. We discuss ECN further in Section 2.1.1.

Honda *et al.* used measurements taken by their tool, TCPExposure, to examine how TCP options are treated by middleboxes [11]. They found instances of TCP options, both known and unknown, being stripped from packets, sequence numbers being translated, and even some port-specific behaviors where options were stripped on a random high port, but not on port 80. Middleboxes along some paths were also found to be very fragile dealing with

Bits	Code Point	Meaning
00	Non-ECT	Not ECN-capable
10 01	ECT	ECN-capable, no congestion
11	CE	Congestion encountered

Table 2.1: ECN code points in the IP header

out-of-order data. Ultimately, they found that at least 25% of the paths seen in the study had a middlebox whose behavior depended on the transport-layer (e.g., TCP) of packets that passed through the middlebox. Not only is this interference detrimental to the validity of the protocol interactions, but it is also difficult to diagnose and makes troubleshooting a complex endeavor.

2.1.1 ECN

ECN [17] is an interesting TCP/IP enhancement worth closer examination as experiences with ECN distill the essence of the middlebox problem. Without ECN, a TCP sender must rely solely on timeouts or lost segments in order to infer the congestion state of a path. With ECN, however, routers assist in congestion control by marking packets as local buffer pressure increases, thereby signaling the sender to reduce her rate before the router will be forced to drop packets.

The ECN protocol relies on a delicate series of cross-layer interactions to function properly: routers must be able to mark packets (at the IP layer), the receiver must echo the congestion mark back to the sender (at the transport layer), and the sender must properly acknowledge and slow down. There are many opportunities for this series of interactions to be disrupted. Not only must the three parties involved (sender, receiver, and routers) properly follow the protocol, but any middleboxes along the path must retain all of the ECN semantics.

At the bit level, the ECN semantics are transmitted via several fields within the IP and TCP headers. The congestion mark is made by setting a pair of bits in the IP header, shown in the middle of the top row of Figure 2.1. The two bit field can take on three different meanings as shown in Table 2.1. When a router wants to mark congestion, and the packet is ECN capable, the router changes the ECN-Capable Transport (ECT) code point to the Congestion Encountered (CE) code point. A receiver that gets a packet with the CE code point set knows that congestion occurred and must tell the sender to slow down. This is done by setting the ECN Echo (ECE) flag in the TCP header of returning acknowledgment packets until the sender gets the message, slows down, and confirms via the Congestion Window Reduced (CWR) flag of the TCP header. Both flags are shown on the left side of the fourth row in Figure 2.2.

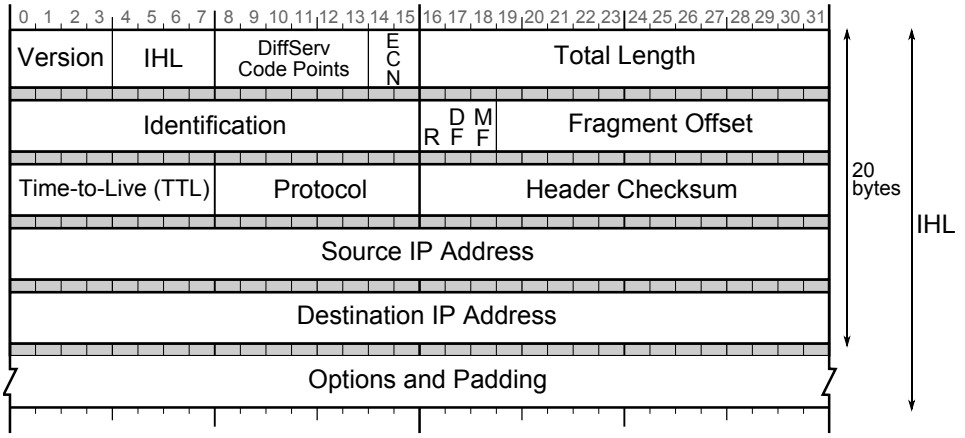

Figure 2.1: Structure of an IPv4 header

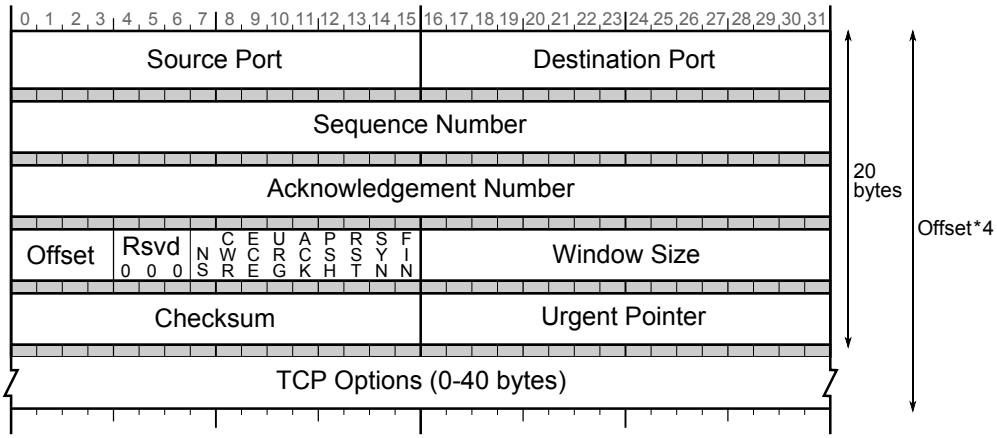

Figure 2.2: Structure of a TCP header

2.1.2 Issues with legacy devices and ECN

It is important to note that all of the bits occupied by these ECN flags had other meanings before ECN was standardized in 2001. The DiffServ and ECN code points fields shown in the top row of Figure 2.1 were originally a single byte known as the type of service (TOS) field. The TCP flag bits shown on the left side of the fourth row of Figure 2.2 were listed as reserved and expected to always be zero. An experimental enhancement to ECN adds semantics to another bit in the TCP header, the nonce sum (NS) bit [18]. Each of these field redefinitions is an opportunity for a legacy middlebox to misinterpret packet header bits and potentially disrupt the ECN interactions.

While the study by Medina *et al.* found issues with ECN-blocking middleboxes, it was

performed in 2004 when ECN was fairly new without widespread implementation. 93% of the servers they tested did not even support ECN. Seven years later, in 2011, Bauer *et al.* revisited ECN readiness in Internet hosts and found that even though many servers were capable of using ECN, a non-trivial number of problems with middleboxes disrupting the ECN fields still existed [12]. The most common problem involved treating the 6-bit Differentiated Services Code Point (DSCP) field and the 2-bit ECN field as the old 8-bit TOS field. Attempts to overwrite or clear the DSCP field resulted in accidentally overwriting the entire byte and destroying the ECN information.

Such overwriting can also impact connection performance. For instance, if any of the congestion signaling bits are inadvertently set when there was not any congestion to begin with, performance will suffer. As stated before in Section 2.1.1, ECN relies on a delicate series of cross-layer interactions to properly communicate when congestion was experienced and when that communication was acted upon. Any spurious overwriting of its field can corrupt the state between the two endpoints, confusing ECN and impacting performance.

2.1.3 ISN translation and SACK

As another example of a protocol being manipulated by a middlebox, some firewalls implement TCP initial sequence number (ISN) randomization to protect hosts behind the device that may be using predictable initial values. Sequence numbers must therefore be translated during the life of the TCP connection. However, in many cases this feature does not properly translate selective acknowledgment (SACK) values and pass the untranslated SACK blocks with translated ISNs. Enabling this feature harms both performance and overall throughput. Documentation from one leading vendor now recommends disabling the module in their firewall [5].

Problems such as a mismatch between SACK blocks and overwritten sequence numbers can be very difficult and time consuming to identify and fix. We encountered this very issue ourselves on our own organization's network. In order to diagnose it, we required a cooperating endpoint to perform low-level comparison between traffic sent and traffic received. A problem such as this is very subtle and requires the keen eye of a trained administrator to recognize and understand the issue.

Instances of SACK mismatches due to middleboxes are found in the academic literature as well. Honda *et al.* gives a general warning about sequence numbers being included in various TCP options due to the fact that they are often inconsistently overwritten by middleboxes [11]. Honda also notes that this issue could even become worse if any of the various proposals to expand the TCP options space [19] are ever adopted. Extending the options across multiple packets could make copies of the sequence numbers even harder for a middlebox to locate and translate. Hesmans *et al.* further discuss the issues with SACK and suggest a change to how out-of-window SACK blocks are interpreted [20].

Protocol Innovation	Reason for Disruption	Issue	Impact	Study
Path MTU Discovery	Legacy, Policy	ICMP blocking	Degraded performance	[16, 21]
IP Options	Legacy, Performance	Blocking, option stripping	Blackholes, poor extensibility	[10, 16]
TCP Options	Legacy, Policy	Option stripping	Poor extensibility	[11]
ECN	Legacy, Misconfiguration	Blackholes, mark concealment, improper congestion signals	Degraded performance, attack congestion control	[12]
SACK	Misconfiguration	Out-of-context SACK numbers	Degraded performance	[11, 20]

Table 2.2: Examples of Middlebox Interference

2.1.4 Negative impact on overall network security

As seen with ECN and other extensions, misconfigurations and legacy behavior in middleboxes can stifle innovations that were designed to add features and make the network more robust. Table 2.2 summarizes some of these examples.

In the case of ECN, for example, it was designed to enhance congestion control in TCP/IP. By integrating state from routers, the network can provide added functionality such as early congestion detection to increase network efficiency and fairness. When middleboxes inadvertently disrupt the interactions of the ECN fields, those gains can be lost and the potential is there to completely break congestion control itself. On paths where congestion marks or echoes are always falsely set, performance will be severely degraded. This can function almost like a Denial of Service (DoS) attack.

In general, the Internet Engineering Task Force (IETF) and various network administrators, especially those with large user bases, are far more reluctant to enable these new extensions when unexpected protocol manipulations are taking place and causing connections to fail. With respect to ECN in particular, the feedback we got from one large content provider was that "we want to enable ECN, but do not because enabling ECN may adversely affect some of our users." [22]

This reluctance induced by a small number of bad paths can negatively impact the overall security and stability of the Internet as our usage and needs evolve, bringing with them the need for new extensions to core network protocols. In this sense, middleboxes and

Section	Name	Description	Examples
2.3	Prevention	Stop middleboxes from tampering	IPsec, tcpcrypt
2.4	Avoidance	Fix middleboxes before issues occur	SIMPLE, APLOMB
2.5	Detection	Detect and potentially fix or workaround	checksums, Tracebox

Table 2.3: Categorized breakdown of the current solution space

other systems that cause these problems are *inadvertently adversarial*. This differs from the typical adversarial model in that while a system is not intentionally malicious, it can and does cause unforeseen problems through its modifications of packet fields.

2.2 Solution Space

We surveyed the current solution space for this problem and found that the various alternatives can be categorized under one of three main styles of approaches: prevention, avoidance, or detection. A summary of our three categories along with pointers to their corresponding sections is shown in Table 2.3. Prevention involves using strong cryptography to stop middleboxes from tampering with packet headers. Avoidance tries to fix the middleboxes themselves before issues arise. The final category, detection, includes solutions that check to see whether any modifications were made to the packet headers in-flight.

2.3 Prevention

A great deal of prior network research has focused on protection in the presence of a motivated attacker with the means to arbitrarily inject or modify packets [23]. This has led to the development of several protocols designed to offer protection from strong adversaries capable of packet modification and injection. Given the strength of adversary in their security models, many of these solutions are often stronger than necessary for our problem, but they can be used to prevent packet header modifications by a middlebox.

2.3.1 Strong cryptography

Internet Protocol Security (IPsec) is a suite of security enhancement protocols that operate at the IP layer [24]. The technology guarantees connectionless integrity and data origin authentication of packets through the use of strong cryptography. One of the more noteworthy components of the suite is the Authentication Header (AH) [25]. The AH is used to ensure the integrity of a packet header and its payload by protecting a collection of fields defined as immutable. It acts as an additional layer on top of IP, so all systems along a path must be able to support it. Drawbacks when applied to our problem include: low adoption, poor interoperability, and lack of a effective and trustful key exchange mechanism. Furthermore, IPsec could not be applied to general Internet paths, i.e., between a user at a coffee shop and each of the servers of websites they may choose to visit.

The addition of message authentication codes (MACs) to TCP packets for authentication and integrity was first implemented by the TCP MD5 Signature option standardized in RFC 2385 [26]. The standard specifies for the inclusion of a 16-byte MD5 digest in the options space of every TCP packet of a connection. The motivation for this standard was to secure long-lived flows that eventually wrapped sequence numbers, allowing for easy packet injection attacks. It mostly applies to Border Gateway Protocol (BGP) connections between BGP peering routers and requires that keys be established through a manual initial setup or some other out-of-band mechanism.

The TCP MD5 standard later evolved into, and was obsoleted by, the more generalized TCP Authentication Option (TCP-AO) described in RFC 5925 [27]. TCP-AO enhances the strength and flexibility of the MACs over that of TCP MD5 by allowing for the use of stronger algorithms. However, it still requires that keys and certain session parameters, e.g., which MAC algorithm to use and whether TCP options are covered, be established manually or by an out-of-band mechanism. It is still mainly used for the same purpose– long-lived BGP connections.

Secure Sockets Layer (SSL), and later Transport Layer Security (TLS), use public key cryptography to authenticate one application to another and establish a session key for data encryption [28]. It operates above TCP, but can be used to encrypt traffic from a variety of applications. The cryptography is strong enough to challenge a motivated and capable adversary and most successful attacks rely on some indirect attack to the encryption, like a man-in-the-middle attack. Since it operates above TCP, it does not protect the IP and TCP packet header fields as they traverse the network, only the application-layer message. This is a major reason that it cannot be applied to this problem, but there are also issues with key management, establishing chains of trust within TCP where they are less easily updated, and computationally expensive public key cryptography.

2.3.2 Opportunistic encryption

Various issues with key distribution and infrastructure have also led to various opportunistic approaches to encryption. With opportunistic approaches, the authentication requirements are weakened meaning that establishing an encrypted session is easy, but no guarantee is made that it is the correct party instead of some man-in-the-middle.

Langley proposes widespread encryption of Internet traffic in order to limit trivial eavesdropping on public networks and click stream monitoring by unfriendly ISPs [29]. The suggested strategy is to cram a Diffie-Hellman exchange into the TCP options space to bootstrap a session key.

Better-than-nothing-security (BTNS) is an unauthenticated mode of IPsec [30]. BTNS uses self-signed keys to avoid the step of having to verify identities. Due to its easy vulnerabil-

ity to man-in-the-middle attacks, the authors recommend combining it with a higher-level authentication mechanism that cooperates with IPsec.

Tcpcrypt is an extension to TCP that was released in 2010 to perform opportunistic encryption of TCP connections [31]. The protocol defines new key exchange primitives called `CRYPT` options for the TCP option space that enable encryption keys to be negotiated directly within TCP. Once a shared session key is negotiated, all information in the headers and the data itself is encrypted and a keyed MAC is used to ensure integrity.

2.3.3 Limitations and Drawbacks

Interoperability

Many of these protocols that provide strong security guarantees share a common theme: strategies that require the network to understand a new protocol or extension yield low rates of adoption. Tcpcrypt, at least, is interoperable due to its use of the options space. If a system does not support Tcpcrypt, it simply ignores the option. A problem with this approach, however, is that options may be dropped or mishandled by any system in-line. This would be an excellent place for our approach to complement Tcpcrypt, as an endpoint wanting to use encryption could ascertain which, if any, of its options were being modified or dropped.

Complexity and overhead

Solutions developed for the strong adversary can certainly be applied to counter misconfigured middleboxes, but due to the security model used in the design, they employ more complexity and overhead than necessary and suffer from compatibility limitations that keep them from being widely used.

Middlebox cooperation

These tamper-prevention solutions are also uncooperative with good middleboxes. Network administrators need to still be able to enforce their corporate policies and, as such, a solution that leaves control information readable would gain much wider acceptance. We believe that implementing a solution that simply adds integrity information to packets rather than complete header encryption would allow for a higher level of interoperability and acceptance. We maintain that interoperability with current network devices could be achieved while still being able to successfully detect modifications and improve performance in the presence of disruptive middleboxes.

2.4 Avoidance

Recently, the research community has paid significant attention to various means of explicitly accommodating middleboxes and thoughtful redesigns of middlebox architectures. The community is well-aware of network administrators' increasing reliance on middleboxes in their networks [32], a market estimated to reach more than $10B by 2016 [33]. This figure

alone is evidence that middleboxes are here to stay and of the value that they provide to networks and their customers.

An early proposal by Walfish *et al.* from 2004 introduced a new architecture that gives all entities globally unique identifiers in a flat namespace while allowing for explicit intermediate packet processing [34]. This ambitious proposal was never used by the community, however.

In the time since, vendors of WAN optimizers have also recognized the problem of middlebox cooperation in traffic modifications and have begun adding their own TCP options, e.g., the TCP Middlebox Option, that request voluntary detection of other middleboxes along a path [35]. The option only helps specific devices from certain vendors that support it; legacy devices will not only not support it but will likely strip it as well. The option also has no end-to-end meaning and is commonly removed from a packet before it reaches its destination.

2.4.1 Software-defined middleboxes

Beginning in 2011, many in the community began to advocate for the application of principles from software-defined networking (SDN) to middlebox architectures [14, 36]. The belief is that ideas and practices from the field of SDN have the potential to reduce the sprawl of standalone, non-cohesive middleboxes and unify control over middlebox operations. Since then, a variety of solutions employing these principles have been developed [3, 37–40].

xOMB (pronounced "zombie") [37] is a modular software-defined middlebox architecture that utilizes commodity hardware and operating systems to implement a middleboxes services framework. While debugging the modules is easier than a standalone middlebox, the framework does not implement any checks for packet modification correctness, so having correctly operating and up-to-date xOMB middleboxes still depends on the skill and attention of the local network administrators.

CoMb [39] is a top-down redesign of middlebox infrastructure that seeks to develop a more open and extensible middlebox platform that will allow for the consolidation of the middleboxes on a network, reducing device sprawl. With a minimal performance overhead, CoMb reduces the number of different devices and different platforms by consolidating middlebox functionality within a single logical controller that can be more centrally managed. A prototype built using the Click modular router [41] showed benefits to the cost of provisioning a new middlebox and reducing the maximum load across the network as the middlebox deployment is adjusted to changing traffic workloads.

SIMPLE (Software-defIned Middlebox PoLicy Enforcement) [40] is an effort to restruc-

ture middlebox processing within the network. Designed to work within the constraints of pre-existing middleboxes and SDN interfaces, SIMPLE requires no changes to a network's current middlebox deployment—only configuration of SDN-enabled switches are required. SIMPLE uses a controller made up of three key modules (ResMgr, DynHandler, and RuleGen) to apply a high-level middlebox policy to a network. Middleboxes are treated like non-adversarial blackboxes and rules for their input-output behaviors are automatically learned by the controller. The authors achieve close to 95% accuracy using their protocol-agnostic approach. SIMPLE's primary benefits are to deployment flexibility and load-balancing efforts. For example, they were able to achieve the same maximum load benefits as CoMB without having to modify or consolidate the middleboxes.

Outsourcing middleboxes
Jingling [38] is a prototype outsourcing architecture where the network forwards data out to external "Feature Providers" that can dynamically adjust to changing traffic loads. The feature providers apply equivalent middlebox functionality to the network's traffic so that the network can eliminate their own local middleboxes, thereby reducing cost and management complexity. This technique allows consolidation of middleboxes from multiple networks under one authority than can theoretically do a better job of configuring and updating the middlebox deployment. The relation to our problem is that this strategy could help eliminate many of the issues we believe we need to address, depending on deployment.

APLOMB [3] is a service to outsource certain types of middlebox processing to the cloud for ease of management. An APLOMB gateway device is installed so that it is logically co-located with an enterprise's gateway router and replaces all of that enterprise's middleboxes. The APLOMB gateway securely tunnels all applicable traffic out to a selected datacenter cloud presence where the middlebox processing is applied to the traffic. An effort is made to reduce the latency and bandwidth inflation penalties involved while still achieving the equivalent functionality of a traditional middlebox.

2.4.2 Ineffectuality
While the schemes presented in this section are laudable, they depend on deployment and use. The authors of SIMPLE even noted in their review of prior work that all of these schemes exhibit significant barriers to adoption and the incentive just is not there for a network to overhaul its entire middlebox deployment. This is the reason they tailored their design to work within the constraints of legacy middleboxes and existing SDN interfaces—to reduce the impact of adoption and make for easier deployment.

Even if wider deployment is achieved, there is still no guarantee with any of these schemes that the problems mentioned in Section 2.1 would be fully eliminated. Each of these schemes only makes debugging easier by consolidating middleboxes where they can be more easily managed than traditional standalone middleboxes with closed interfaces. The

frameworks themselves do not implement any validation for protocol correctness on packet modifications, so misconfigurations and non-standard behaviors will still be possible.

We believe that there is an even more fundamental inhibitor to the efficacy of these schemes in solving our problem: incentives and (lack of) policy. All of these software-defined management approaches are confined to single administrative domains—domains which may or may not have the incentive or policy to convert its middlebox deployment to a SDN-based approach. Therefore, TCPs in the wild must still contend with a wide variety of middleboxes. We believe that even if many subnetworks begin to adopt and implement one or more of these schemes, and even implement a validity checker on top of them, there is still great value in an end-to-end solution that will work over all paths, particularly ones with legacy middlebox deployments.

2.5 Detection

The networking community has paid a great deal of attention to detecting modifications to packet headers, but traditionally the problem has only been simple transmission errors. More recently, a greater amount of attention has been paid to the types of modifications introduced by middleboxes.

2.5.1 Simple checksums

Protection against transmission errors was considered during the design of the Internet protocol suite and is built into the stack via various link-layer mechanisms and network and transport layer checksums.

Two of the most commonly used link-layer protocols both employ a cyclic redundancy check (CRC) to detect corrupted frames: Institute of Electrical and Electronics Engineers (IEEE) 802.3 (Ethernet) [42] and IEEE 802.11 (WiFi) [43]. Also, not only does the 802.11 protocol family include a check sequence in each frame, but acknowledgment (ACK) frames are used to confirm a receiver's proper receipt of a frame. The absence of one after a certain period of time is the signal for the other end to retransmit [43]. These checks can detect modifications on a single link, but they have no end-to-end significance.

The IP, TCP, and UDP protocols all include a checksum as well that is 16-bits in length. The Internet checksum algorithm that is used is weaker than a CRC, but can be efficiently implemented with very fast binary operations. The algorithm computes a one's complement sum of 16-bit chunks of data that are to be included in the checksum [44]. The IP checksum only covers the IP header, while the TCP and UDP checksums cover a pseudo-header that includes some IP header fields, the TCP/UDP header fields, and the packet data.

The IP checksum does not apply well to the solution we seek because it does not cover any transport layer fields. Furthermore, IP checksums are not end-to-end as they cover fields

that change in transit such as the time-to-live (TTL) field. This fact forces the checksum to be rewritten at each hop, meaning that errors occurring within those intermediate systems, the middleboxes that introduce the errors, will not be detected. Another issue is that IPv6, the anticipated eventual replacement for IPv4, does not even include a checksum.

Transport-layer checksums do have end-to-end significance, but it is conditional upon transport-layer fields not needing to be modified by a middlebox. Since a correct checksum is required for an endpoint to accept a segment, middleboxes must recompute the checksum anytime they make a change. There is then no way for either endpoint to know whether the checksum received is the same as the original checksum. There is also no way for a sender to know if the received checksum was even correct, let alone the same as the original.

Stone and Partridge note this deficiency of a feedback mechanism and suggest the addition of a new Internet Control Message Protocol (ICMP) parameter to alert the sender of a failed checksum [45]. A problem with this out-of-band method is the reliance on the availability of a secondary communications channel, ICMP, which is commonly blocked by middleboxes as noted in Section 2.1. Relying on ICMP may therefore negate the ability to communicate integrity, especially on those networks and paths most likely to modify packets and fail integrity.

Another problem with checksums in general is the lack of granularity down to individual header fields. All of the checksums discussed in this section only provide a binary answer to whether the header as a whole has been modified. A change in any single field will cause the whole checksum to fail to match, which will not give a TCP the full information it needs to reason about the correctness of a path.

2.5.2 Application layer solutions

Some solutions also exist at the application layer: Switzerland [46] by the Electronic Frontier Foundation (EFF) and Netalyzr [4] by the International Computer Science Institute (ICSI). Both of these solutions are application-layer approaches to check if packets are being altered by middleboxes and were primarily developed as network neutrality analysis tools.

Switzerland works by comparing packets sent against packets received at the other end by cataloging mini-hashes on a third-party server. This approach can be used to detect in-network modifications in a similar fashion to the proposed TCP-based integrity checks, however it is not integrated into TCP, requires availability of a third-party, and is not very widely used.

Netalyzr is a Java-based applet that performs a multitude of checks between the Java client and a set of back-end servers to spot specific modifications to traffic and aid in network

diagnostics. Its primary drawback for our specific problem is that the Java security restrictions severely limit the lower-level networking tasks that can be performed. For example, they cannot view sent or received TCP sequence numbers at the client within the limits of their implementation.

A key limiting factor to both of these options is that both ends of a connection must be running the program, which leads to low rates of adoption. This also means that only certain paths, such as the one between a host and the Netalyzr servers, can be tested for modifications. We believe that by extending down into the network stack, a TCP-based technique could lead to more pervasive adoption and the ability to test for modifications with any system on the Internet. This would also have a large impact on the Internet measurement community, as any end host could be considered a cooperating endpoint for testing.

2.5.3 Tracebox

Tracebox [47] is a tool that can detect in-path packet header modifications under certain conditions. It is marketed as an extension to traceroute [48] that works by sending TTL-limited TCP probes and examining ICMP quotations from the ICMP TTL-exceeded messages that come back when the packet expires.

The RFC that defines the ICMP protocol says that TTL-exceeded messages should include a quote of the IP header and the first 8 bytes of the IP payload of the packet when it expired [49] (28 bytes in total). This allows the sender of the TTL-limited packet to obtain feedback and get a view of the packet's state along a path. For TCP packets, observing only the first 8 bytes of the IP payload implies visibility into only the sequence and acknowledgment numbers. This makes it impossible to spot changes to other fields of the TCP header, and in particular the important options space.

The creators of Tracebox noticed that RFC 1812 [50] recommends a new quoting behavior for ICMP TTL-exceeded messages. The new recommended behavior is to quote as much of the expired IP packet as possible back to the sender. This allows the sender of the packet to observe packet headers for each TTL-exceeded message and find differences from how the packet was originated. The authors found that many newer routers actually implement this RFC 1812 behavior and wrote Tracebox as a means of automating the differencing of the IP and TCP headers.

The benefits of the methodology used by Tracebox are that information is learned not only about what fields were changed, but also the values that they were changed to and also where along the path the change occurred. Determining which hop along a path induced a modification is going to be difficult for any purely end-to-end-centric strategy. There are also fewer restrictions on which types of packets can be checked; use of the tool is not

limited to synchronize (SYN) packets as any packet can have its TTL lowered. Another positive of the methodology is that so-called "full-quote" routers, i.e., routers that follow the quoting behavior recommended by RFC 1812, are becoming more and more common on the Internet. In a small measurement study using 72 PlanetLab nodes, they found that 80% of the 360k paths examined contained at least one full-quote router.

While Tracebox is useful, it does have certain limitations. For one, Tracebox assumes a completely cooperative environment. Routers have to be trusted to properly quote the packet, not tamper with any other ICMP TTL-exceeded messages being forwarded through them, and network policy restrictions have to allow the ICMP messages to make it back to the sender. As seen with Path MTU Discovery (PMTUD), neither an open policy for ICMP nor even being properly configured to forward ICMP can be taken for granted [16, 21]. Furthermore, if a client is stuck using an unfriendly Internet Service Provider (ISP), it is trivial for that provider to limit the effectiveness of the technique or induce false readings. Other issues include lack of visibility to changes occurring in the penultimate hop (because the final hop does not expire and quote the packet), and a reduction in location accuracy as fewer hops along the path provide quotes.

2.5.4 Common Limitations

A common issue among detection-based techniques is that they do not work with TCP so that it would be able to dynamically adapt to middlebox behaviors. Checksums are recognized by TCP, but do not provide the type of information we need. Application-layer or out-of- band solutions are not integrated with TCP and do not have a way to provide the information to TCP. Furthermore, any attempt to implement this bridge would only result in a partial solution: the reliance on availability of an application or out-of-band mechanism severely restricts the number of paths for which TCP would have the additional information. In order to be fully cooperative with middleboxes, TCP must have information about changes to packet headers so that it can reason about protocol correctness on its own and adjust its behavior to best match the header modification conditions along a path.

CHAPTER 3:
Requirements

3.1 A TCP-based Integrity Check

In order to solve our problem as described in Section 1.2, we propose the development of an in-band TCP-based integrity check to detect packet header modifications that occur along a path. The detection solution should endow a pair of endpoints with the ability for each to determine whether their packet headers were modified in transit. Our solution should be incrementally deployable and require no support from transit devices to ensure interoperability.

Through this work, we aim to address an important class of problems due to misconfigured, non-standards conforming, or legacy in-path network elements and endow endpoints with the necessary awareness so they can take some appropriate action, such as disabling an incompatible option or extension.

3.1.1 Security model

In our security model, we operate under the assumption that in-path network elements are not actively malicious. In other words, we shall make no guarantees of protection from strong adversaries and the range of attacks they pose, e.g., man-in-the-middle attacks, cryptanalysis, or side-channel attacks. We also do not strive to provide the endpoints with a means of confidentiality from these devices. Not only would the provision of these guarantees further constrain our solution space, but as we stated in Section 2.3.3, would make our solution less cooperative with good middleboxes and hurt our likelihood of achieving wide deployment and acceptance within the community.

Therefore, we assume the presence of an inadvertent adversary, a network element somewhere along a path that is not actively malicious but is inadvertently corrupting critical packet semantics. By using this adversarial model, we hope to achieve more desirable interoperability properties in our solution, namely greater flexibility and incremental deployability. As we showed in Section 2.2, traits such as these are typically sacrificed when strong cryptography is needed. Striking a balance here is often very difficult but we believe our tailored security model will allow our solution to excel in this problem space.

3.2 Design Requirements

Upon surveying the solution space and the current state-of-the-art, we find many limitations with each possible solution that we hope to overcome in our design. In order to capture these limitations and show how we would like to advance the solutions space, we define the following set of design requirements:

- **In-band**: The solution should utilize the same communications channel as the traffic being measured. Many paths block out-of-band traffic, e.g., ICMP, or treat it differently. Having both the detection and feedback mechanisms in-band will maximize our detection rate.
- **Minimal overhead**: The design should be efficient and lightweight, resulting in a limited amount of overhead in terms of computation, communication, and round trip times (RTTs).
- **Symmetric feedback**: It is important that hosts at each end of a connection know whether and how their packets were modified in flight.
- **Incrementally deployable**: The solution should be incrementally deployable and not require updates to in-network elements. The design should also not interfere with end-hosts that have not yet been upgraded, i.e., if a connection completes when neither end is using our solution, it should still do so if either or both ends are using our solution.
- **Improves TCP**: The design should endow endpoints with the necessary awareness so that they can take some appropriate action, such as disabling a non-compatible option or extension in order to improve performance. To maximize utility, it should be easily integrated into host protocol stacks.
- **Middlebox cooperative**: The solution should not impede good middleboxes from making expected and desired changes to packet headers.
- **End-to-end**: Paths exhibiting modifications are often the same paths most likely to block or strip any new diagnostic functionality. The diagnostic should be properly communicated end-to-end.
- **Granular**: Endpoints should be able to determine which packet header fields were changed.

One key to developing a *novel solution* that improves upon those currently available is the fresh point-of-view provided by our security model. Much of the prior network research has focused on the edges of the spectrum: protecting integrity from either transmission errors or from strong adversaries. When operating under the model of the inadvertent adversary, the solutions developed by those works are either too weak to be useful or make too many sacrifices in pursuit of strong cryptographic assurances. Our approach admits new possible solutions that have the advantage of interoperability with current devices, while still being able to reliably detect common middlebox-induced modifications.

3.3 Comparison with Solution Space

Before explaining our resulting designs, we place our solution in the context of the integrity and middlebox cooperation schemes described in Section 2.2. Table 3.1 describes the degree to which these relevant prior works meet the corresponding design objective. Checksums are featured in the table due to their widespread use within the current proto-

cols. The other three items in the table were chosen as the best exemplar from each of the three groupings under which we categorized the solutions space.

Scheme	In-band	Minimal overhead	Symm. feedback	Incrm. depl.	Impr. TCP	Mdl. coop.	End-to-end	Granular
Checksums	●	●	○	●	○	●	○	○
Tcpcrypt	●	◔	◑	◕	○	◔	◑	○
Tracebox	○	◔	◑	◕	○	◕	◔	◕
SIMPLE	○	◔	○	◔	◕	●	◔	●

Table 3.1: Summary of Related Work

In Table 3.1, Harvey Balls are used to represent the degree to which each integrity or middlebox cooperation scheme meets each of our design criteria. A full ball, ●, means the scheme fully met that criterion. An empty ball, ○, means that the scheme failed to meet that criterion. Other levels imply partial meeting of the criterion with possible caveats. The following lists explain the reasoning behind each decision:

Checksums:

- **In-band**: They are carried within both TCP and IP.
- **Minimal overhead**: The algorithm is lightweight, and only requires extra transmissions when it fails to match.
- **Symmetric feedback**: There is no feedback mechanism.
- **Incrementally deployable**: They are already deployed and boxes understand them.
- **Improves TCP**: Does not help improve TCP under the presence of disruptive packet header modifications.
- **Middlebox cooperative**: Middleboxes can make any changes as long as they recompute the checksum.
- **End-to-end**: Must be overwritten if any packet header modifications are made.
- **Granular**: No granularity to individual fields.

Tcpcrypt:

- **In-band**: Operates fully within TCP.
- **Minimal overhead**: Encrypts opportunistically but uses strong crypto and uses a large portion of the options space.
- **Symmetric feedback**: Communicates through TCP options, but fails to give status in certain situations.
- **Incrementally deployable**: They are already deployed and boxes understand them.
- **Improves TCP**: Prevents changes, but cannot help two endpoints optimize parameters for a disruptive path.

- **Middlebox cooperative**: Middleboxes cannot change packet headers once the connection is encrypted.
- **End-to-end**: Will not work on all paths because it is susceptible to having its options stripped.
- **Granular**: No granularity to individual fields.

Tracebox:

- **In-band**: Relies heavily on ICMP messages.
- **Minimal overhead**: Requires successively fractional RTTs similar to `traceroute`.
- **Symmetric feedback**: Host being probed does not learn any information.
- **Incrementally deployable**: For full functionality, routers need to be upgraded to support RFC 1812-style packet quoting.
- **Improves TCP**: Does not give any information to the TCP stack.
- **Middlebox cooperative**: Middleboxes can still make any changes, but only checks one path.
- **End-to-end**: Cannot determine modifications made by penultimate hop.
- **Granular**: Gives granularity when router quotes full length of headers.

SIMPLE:

- **In-band**: Requires SDN infrastructure
- **Minimal overhead**: Requires testing for correctness of protocol behavior.
- **Symmetric feedback**: Information is retained by the network operator.
- **Incrementally deployable**: All middleboxes along a path must be upgraded in order to realize benefits.
- **Improves TCP**: Correctness-testing in the SIMPLE architecture would help avoid problems.
- **Middlebox cooperative**: Uses dynamic learning module to work with any middleboxes.
- **End-to-end**: Has no effect on middleboxes outside the owner's administrative domain.
- **Granular**: Learns individual field modifications.

In the context of these existing and proposed integrity and middlebox cooperation schemes, we find that our design meets our objectives and represents a unique point in the design space. In the following chapters, we describe detection schemes that will fully meet each of the above design criteria.

CHAPTER 4:
Transmitting Integrity

In order to provide end hosts with the power to detect in-path modifications to their traffic, we need to address the fundamental problem of how to communicate an integrity check of the packet header fields and any related status information. In Section 4.1, we examine various methods and analyze the extent to which each satisfies the requirements listed in Chapter 3. After settling on a strategy that is better suited to the problem than the others, we analyze the design considerations under that strategy in Section 4.2. Finally, we enumerate an array of potential design alternatives, along with their pros and cons, in the variants described in Sections 4.3 through 4.8.

4.1 Methods for transmitting integrity

4.1.1 Application layer

An application layer methodology is easy and quick to implement and use. We are not bound by TCP in our capacity to make use of strong hashing functions and entire packets could be echoed back to their senders to facilitate easy comparison. There are many opportunities at the application-layer to create an ideal design, but a critical drawback for us is the low rate of adoption. We require a technique that can become a natural extension to one of the lower-layer protocols and gain widespread use. Furthermore, we do not want a specific application to have to be running on each host in order to examine a path between endpoints. Problematic middleboxes can be found on a multitude of paths within the Internet and we desire the capability to diagnose any related issues on that path whether or not our destination is running a specific application-layer server program.

4.1.2 ICMP

Upon initial observation, ICMP is a natural vehicle for the integrity information we need to communicate. The protocol itself is designed to facilitate diagnosis and transmit error and control information about IP packets. We gave initial consideration to using new types of ICMP packets to carry integrity and generated the following scheme ideas:

ICMP trigger When a sender is concerned about tampering by a middlebox, the system sends an ICMP message to the destination host to trigger an echo. After receipt of the trigger, the destination echoes back the next packet that it receives from that source host. When the sender receives the echoed packet, a 1:1 comparison can be performed between the two versions of the packet.

ICMP error If another integrity check service is in use, echo any packets back to the sender that fail the integrity checks.

ICMP hybrid 1 The same as the "ICMP error" method, but only echo the failed packet if an ICMP trigger message has been received from the source.

ICMP hybrid 2 The same as "ICMP error", but only include the byte offset where the error occurred in the original ICMP error notice. Then, listen for a trigger message from the source to see if that system wants the full packet echoed back.

An issue shared by all of these approaches is that they all require the availability of an out-of-band mechanism. Given the issues with Path MTU discovery [16, 21], we would expect to encounter a sizable portion of paths where our ICMP messages would be blocked. Therefore, the number of paths that support detection with ICMP would be a subset of the in-transport results. Since our targets for detection are paths that experience issues due to misconfigured and non-standard middleboxes, we cannot afford to miss results on these problematic paths. There is no additional benefit to ICMP over an in-band method with respect to having widespread ability to test paths.

4.1.3 TCP

A benefit of working within TCP is that all integrity transmissions occur in-band. If a TCP port is open and a connection taking place, we should be able to transmit our integrity bits if we can integrate them into TCP.

The TCP options space is commonly used by extensions and Honda *et al.* showed that many unknown options are left intact for both SYN and data packets [11]. With that in mind, one alternative could be to allow for the sender of any TCP packet to include an echo trigger within the TCP options of that packet. Upon receipt of a packet with the trigger in the options, an ACK message is generated for that packet with a quote of as much of the headers as possible in the options of the ACK message. This would allow the sender to perform a comparison of the two versions of the headers. Some type of compression strategy may be needed, however, since there would be more headers than could fit within the remaining space for TCP options.

Another method we devised for transmitting integrity information is to include it in the fixed-length fields of the headers. We create space by *overloading* certain fields that allow for a degree of flexibility in their values, such as the IP identification (IPID) field and the initial sequence number field. This method limits us to a small transmission capacity for

our information, but yields many other nice properties such as good interoperability, testing availability, no additional RTTs or bandwidth consumed, and symmetric notification is easy since the concept of a connection is already well-defined. We continue our analysis of design alternatives by looking more closely at these TCP-based methods.

4.2 TCP design considerations

4.2.1 Fields for transmitting the integrity information

To implement the proposed TCP-based integrity check, the two systems communicating in a TCP session need to transmit to each other a representation of the packet header states as each side sees them. To achieve symmetric notification, they must then be able to compare the state representation as they see it upon receipt with the representation seen by the sender. The primary issue will be to find which fields can be used to carry the integrity check.

IP and TCP options space

An examination of previously proposed protocol extensions suggests places to avoid as well as many opportunities. The obvious place to begin looking for space in the headers is the options fields. The consensus here is mixed. While TCP options may be acceptable and commonly used, IP options, as Fonseca *et al.* put it, are not an option [10]. This is due to IP options not being well supported in the Internet. Many devices aim to minimize processing time when routing and forwarding IP packets by ignoring or stripping IP options and only examining the first twenty bytes of the header.

TCP options tend to have much more flexibility, do not impose a performance penalty, and carry with them fewer traversal issues. Again, prior measurement studies found that TCP options, even non-standard ones, are often maintained by middleboxes during transit—Honda *et al.* found that in the worst case, 80% of paths maintained unknown options [11]. Use of the options are common; the two TCP-based security schemes described in Section 2.2 use their own TCP options for extra space to carry key exchanges and hash values.

Problems with using the options include dealing with the paths were options are stripped and overcrowding. Even if the majority of paths maintain unknown options, it still only takes a small fraction of paths that do not in order to prevent or demotivate a new protocol from being deployed. In terms of overcrowding, Ramaiah takes note of the various proposed extensions competing for TCP option space and finds that the originally designated 40 bytes of options are no longer able to meet current demands [19]. When the various proposals are considered, the options space is already overcrowded and demand continues to grow.

URG pointer

Some of the workarounds described by Ramaiah suggest more possibilities such as repurposing of the TCP urgent pointer. The work suggests that the field's use has largely

declined, but repurposing it is likely suboptimal for our needs given that it has prior semantics attached to it and re-purposing it may cause odd behavior with legacy devices. We intend to check this assumption in later work.

4.2.2 Offset checksums

Another workaround suggested in the work by Ramaiah is the interesting notion of deliberately offset checksums. The idea is to send multiple segments covering the same sequence number space, with all but one copy having a bad checksum by design. This solution has nice interoperability properties because any receiver that does not understand the messages will just think they are corrupt and gracefully drop them. Downsides to this option include the excess bandwidth consumed, and the fact that this type of behavior may likely trigger intrusion detection system (IDS) alerts. This idea also relies on the proper behavior of endpoints and in-network elements with respect to bad checksums—that they will gracefully ignore them.

IPID

Another area of promise are fields that are expected to be able to hold any random number. There are two such fields in the IP and TCP headers: the IPID and the ISNs, respectively. The IPID field is a 16-bit field in the IP header that, in the event of fragmentation, is used to differentiate fragments of one packet from another. It does not matter what the value is as long as it is unique to each packet when fragmentation occurs. When there is no fragmentation, the value of the field is of no use in terms of the IP protocol.

Initial sequence numbers

Initial sequence numbers provide a single opportunity for each end host to send 32 bits of information to the other at the time a connection is opened. Hosts can choose any number they like to use as the first sequence number of a connection, but it should be random and unpredictable so as to prevent spoofing and injection attacks. As long as the randomness is reasonably preserved, ISNs may be another avenue through which information can be sent. Just as SYN cookies [51] have previously defined semantics for the ISN value, it should be possible to create a definition of our own. To then leverage the benefits of SYN cookies as well, a fall-back mechanism can be employed in the kernel to switch over to them as server resources become scarce [52].

4.2.3 Possibilities from Network Steganography

We find it may also be of use to more closely examine the field of network steganography. The goal of network steganography is to locate covert channels within the protocol headers or timing mechanisms to transmit hidden data. Usually these steganographic techniques are developed under the assumption that two end hosts and their network stacks are in collaboration and discretely pass data between each other without systems in between noticing or modifying their data. It is an interesting connection to our work because the goal somewhat

parallels our own. We wish to pass the integrity check data through a side channel as well, but do not necessarily bear the requirement that it be covert even though it might be useful to prevent middleboxes from touching it.

An examination of some steganography literature validates our field ideas. Cole uses the IPID field as well as ISNs to pass data [53], Bennet makes use of deliberately erroneous checksums [54], and Luo *et al.* encode hidden data by partially acknowledging smaller pieces of data at a time and having the other end track the amount of bytes acknowledged with each response [55].

While there may be overlap in field usage ideas between our work and that of network steganography, we emphasize that our goals are fundamentally different. We aim to locate fields that will be most compatible with middleboxes. We are not trying to evade detection or design covert communication channels.

4.2.4 Protection coverage

Another design consideration that must be made is to determine which fields should be protected by the integrity check. The RFCs for IPsec define specific fields in the IP and TCP headers as either *mutable* or *immutable*. We take a similar approach with this work and only compute the check across fields deemed immutable, that a middlebox should not modify. Of course, the DSCP and TTL bits in the headers will be mutable, but the check must have the ability to ignore integrity on the common flow identifier fields such as IP addresses and TCP ports. Since NAT devices are prevalent and expected to change these fields, the check has to allow for the option of detecting NAT or ignoring and traversing it.

4.3 TCP design variants

For each subsequent variant section, we define the following key properties:

Throughout connection

We can either protect just the 3-way handshake or the entire connection. Detecting some modifications requires examining a full connection. For example:

- A middlebox, M, modifies initial TCP sequence numbers, but fails to also update SACK blocks into new sequence number window. The SACK blocks will not appear until after the handshake.
- M shrinks the TCP receive window in order to throttle the connection between A and B. This could happen at any point during the connection.

Diagnostic mode

Some of the variants are not able to coincide with a connection for an application, e.g., www, ssh. If so, they will be marked as "diagnostic only," meaning that no application data

should be sent through the connections used by that variant. It can still operate with any open TCP port on a server, but the connection that is created only serves the purpose of checking packet header integrity, after which it is closed.

The ability for a pair of endpoints to create a diagnostic connection creates further issues:

- How does each endpoint detect that a given connection is diagnostic?
- How do you prevent *M* from determining the same thing? If *M* knows which connections are diagnostic, it can adjust its behavior for only those connections.

Fields used
This item lists the fields within the TCP or IP headers that each variant uses in the transmission of integrity or status information.

Raises bar on *M*
At this point we are only focused on communicating integrity information. Chapter 5 takes a deeper look at protecting the integrity and what can be done to stop or at least discourage the information from being modified by the middleboxes we target. For purposes of comparison with the design variants in that chapter, we include a quantification of the level of protection afforded by each variant.

4.3.1 Notation

The diagrams of each variant within this document will show information passing between two hosts, *A* and *B*. The information itself will be consistently described in the format shown in Figure 4.1.

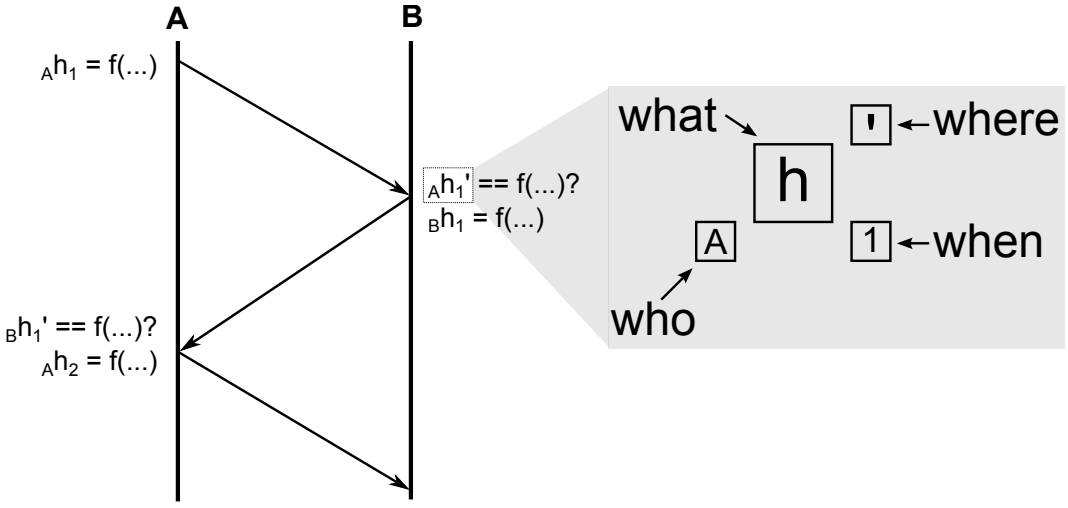

Figure 4.1: Standard notation for variant diagrams

What:	The type of information (e.g., $salt$, h, etc.)
Who:	The party that originated the information (either A or B)
Where:	A prime symbol ($\prime$) here indicates that this is the value of the information after having transited the network. In other words, this information may have been modified by M.
When:	A number n to indicate that this is the n^{th} piece of information of similar type from the same origin. If not present, then it means the information is only sent once from that origin.

Also discussed is $f()$, a publicly known hashing function that converts field state representations to hash values.

4.4 Variant 1: Opportunistic HICCUPS

In this variant, A and B each embed a hash and salt value in their SYN and SYN-ACK, respectively.

Throughout Connection:	No, handshake only
Diagnostic Mode:	None
Fields Used:	Initial Sequence Number (ISN) IPID (on first packet)
Raises Bar on M:	Not really. M must recalculate 2 hash values and at most store 1 packet header for up to half of an RTT.

4.4.1 Detailed Description

The opportunistic variant of HICCUPS has the ability to inform both parties in a TCP connection if their packets were unmodified, without requiring a special diagnostic connection or an extra RTT. This is important for high-performance applications that cannot afford any added delays. The packets exchanged as part of this check look no different to the network than any other similar packets, the only difference is that they have an ISN and IPID that have special meaning. The opportunistic check was designed to operate as part of a normal connection between two hosts that may or may not be using HICCUPS.

An example timeline of the opportunistic check between two hosts A and B is shown in Figure 4.2, where A initiates the TCP active open. When A sends the first packet, it must include a random string (the salt) and the result of the function of that packet's fields and salt value. The salt value is placed in the IPID field and the output from the function is placed in the ISN.

We define the following:

$$\begin{aligned} {}_{A}salt &\leftarrow rand() \\ {}_{A}h &= f(fields_{\texttt{SYN}}, {}_{A}salt) \end{aligned}$$

The function $f()$ is public, e.g., a known hash function. This allows the host at the other end of the connection to compute ${}_{A}h'$ using the standardized function and the fields and salt values of the packet from A as seen by B. If there were no modifications, then ${}_{A}h' == {}_{A}h$. Should they match, B can say that A's packet was unmodified.

At this point, B generates its own salt and ISN value for the returning SYN-ACK packet. If the checks by B pass, then it should incorporate a way for A to know that they passed as well. This can be done by including something known to A in the function input. The fields from the SYN packet can be combined with the fields from the SYN-ACK packet in

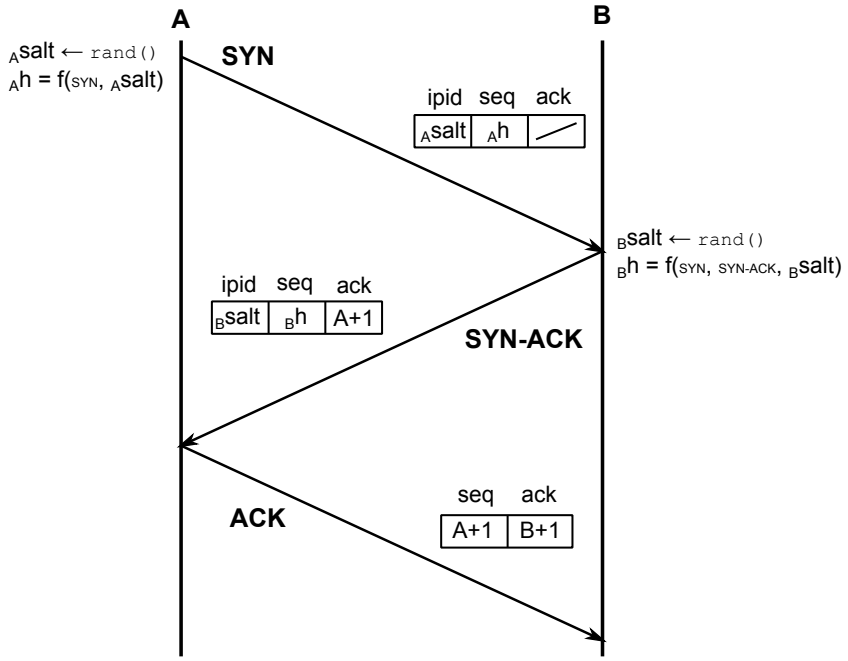

Figure 4.2: Timing diagram with no modifications

calculation of the sequence number used in the SYN-ACK:

$$\begin{aligned} {}_B salt &\leftarrow rand() \\ {}_B h &= f(fields_{\mathrm{SYN}}, fields_{\mathrm{SYNACK}}, {}_B salt) \end{aligned}$$

Should the check fail at B, it could inform A by leaving out the $fields_{\mathrm{SYN}}$ input from $f()$. This would yield the following instead:

$${}_B h = f(fields_{\mathrm{SYNACK}}, {}_B salt)$$

When A receives the SYN-ACK reply from B, it must check the packet's sequence number against both possibilities. The function must be computed using each of the two input combinations B may have used above and determine whether either of them match the sequence number it sees. Table 4.1 lists all possible outcomes at each end of the exchange.

4.4.2 Faking Integrity

In order for M to fool A and B into thinking that no modifications were made, it must simply recalculate ${}_A h$ and ${}_B h$ after performing its modifications. M does not need to regenerate salt values; it can reuse the ones chosen by A and B. Finally, M must be able to store the SYN

At host B after receiving SYN:

${}_Ah == f(fields_{\texttt{SYN}}, {}_Asalt)$	SYN unmodified
else	SYN modified or A not capable

At host A after receiving SYN-ACK:

${}_Bh ==$ $f(fields_{\texttt{SYN}}, fields_{\texttt{SYNACK}}, {}_Bsalt)$	SYN and SYN-ACK unmodified
${}_Bh == f(fields_{\texttt{SYNACK}}, {}_Bsalt)$	SYN modified but SYN-ACK not
else	SYN-ACK modified or B not capable

Table 4.1: Possible outcomes of the opportunistic check

fields until it can calculate ${}_Bh$. At most, this will be until it sees the SYN-ACK return from B. Figure 4.3 summarizes this process.

4.4.3 Pros

- Interoperable, incrementally deployable

4.4.4 Cons

- Does not protect the entire connection
- Can be disabled by overwriting initial sequence numbers
- Bob cannot distinguish between Alice not being HICCUPS-capable and a modified field

4.4.5 Thoughts and Status

Simple; easy to understand and implement. A good first step and proof-of-concept. Implementations of this variant currently exist in the CMAND repo in both kernel and user space versions. See Chapter 6 for more details.

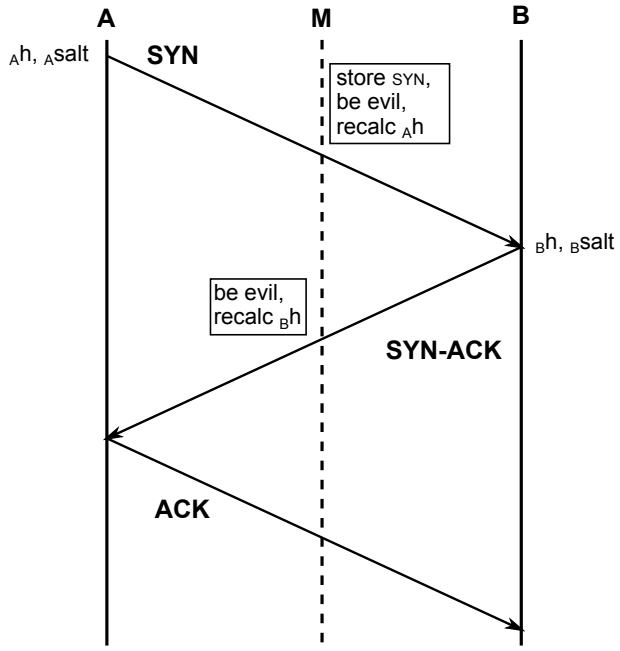

Figure 4.3: Necessary actions to fool A and B

4.5 Variant 2: Offset Sequence Numbers

In this variant, hash values are added to the sequence numbers in each direction.

Throughout Connection:	Yes
Diagnostic Mode:	Yes, diagnostic only *Mode Hidden?* Reasonably so
Fields Used:	SEQ and ACK
Raises Bar on *M*:	Not really. *M* has to guess that the connection is in diagnostic mode, recalculate 2 hash values, and store a sequence number for up to half of an RTT.

4.5.1 Detailed Description

This variant performs additive increases to the sequence number of each packet in the connection, with that increase being expected by the other HICCUPS-capable end-host of the connection. The result of the public function, $f()$, is added to the sequence number at the completion of the handshake. This new sequence number will be outside the receive window of the opposite host, forcing a duplicate ACK from a non-HICCUPS host.

The example timeline in Figure 4.4 is a demonstration of the offset sequence number variant as carried out between two HICCUPS-capable hosts. After *A* completes the handshake, it immediately sends the next packet where the sequence number is $f()$ more than before.

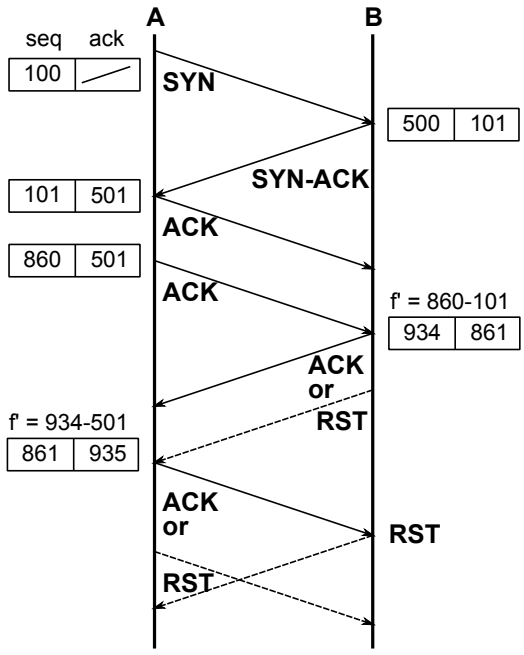

Figure 4.4: Two HICCUPS hosts (no mods)

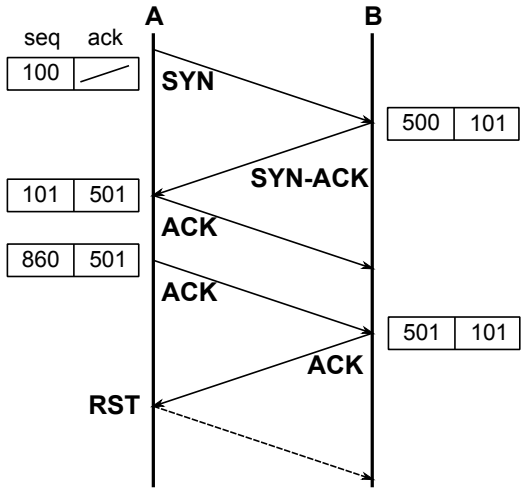

Figure 4.5: B not HICCUPS host

In this example, $f()$ was 759. Upon receipt, *B* can recompute $f()$ from the values of the packet and check to see if it matches the difference in sequence numbers. If it does, *B* does the same thing as *A* and sends a packet with an additively increased sequence number. If it does not, then *B* can send a RST packet to end the connection. *A* then does a similar check in response.

Figure 4.5 shows the timeline where *B* is not using HICCUPS. If *B* replies with a DupACK, we know it must be plain TCP and does not understand our offset sequence number. Thus, we cannot get an integrity check out of this host. Something to note here is that some systems will not respond with a DupACK unless the difference in sequence number is greater than the host's receive window. This can be handled by adding a value to $f()$.

The primary benefit of this variant is that it enables each end of the connection to distinguish between a failed integrity check and a host not using HICCUPS. This eliminates the ambiguity present in the opportunistic variant from Section 4.4. In order to accomplish this, the connection is completely used as a diagnostic connection that does not handle any application data. It should be possible, however, to begin a connection using the opportunistic check and then switch to this variant if there was an issue with the results of the first check.

4.5.2 Faking Integrity

In order for *M* to fool *A* and *B* into thinking that no modifications were made, it must first recognize that a diagnostic connection is taking place and then overwrite the sequence numbers with adjusted values.

Recognization of the mode is a tricky issue. The packets will appear to be out-of-order to most systems, and the check must match up to detect the mode. Granted, *M* could match up the check the same as *B* can, but either way, we introduce a little probability into *M*'s decision.

Once *M* makes the decision that a given connection is performing the offset sequence numbers check, it must perform the actions as shown in Figure 4.6 in order to fool *A* and *B* into thinking that their connection has integrity. This includes saving a 32-bit sequence number for up to half of an RTT and recalculating two hashes.

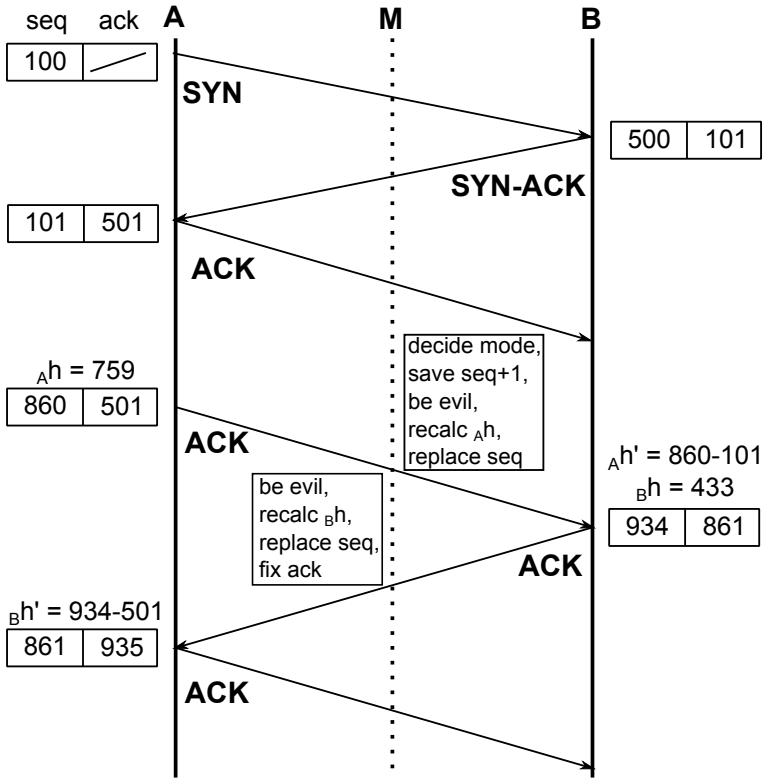

Figure 4.6: Necessary actions to fool *A* and *B*

4.5.3 Pros

- Don't change ISNs, so don't need salt in IPID
- Should get through systems that securely randomize initial sequence numbers since we only care about the deltas

4.5.4 Cons

- Essentially injecting junk into the network which may have unintended consequences
- Some systems may react poorly to the out-of-window sequence number

4.5.5 Thoughts and Status

We do not know what to expect from the network as a whole in handling the offset sequence numbers. This variant essentially involves injecting junk into the network and hoping it is transited properly. Probably not useful on its own.

However, a really big key here is the design facet that it is one way of successfully distinguishing between a HICCUPS host and a non-HICCUPS host without marking up the packets in a manner that may introduce incompatibilities.

4.6 Variant 3: Probabilistic Hashes

In this variant, smaller hashes bounce back and forth between A and B. The result is probabilistic over many trials throughout the connection.

Throughout Connection:	Yes
Diagnostic Mode:	None
Fields Used:	IPID
Raises Bar on M:	No. Per packet it sees, M only has to recalc 1 hash and store 1 hash for up to half an RTT.

4.6.1 Detailed Description

If we allow the hashes to be really small (for instance a single byte), we can squeeze two of them into the IPID fields of every packet in a connection. Obviously with such a small range of outputs for $f()$ we should expect a fair amount of collisions. However, if we do these checks on each packet over the life of a connection, the probability of all of them being collisions becomes very small. This is akin to the way the ECN nonce [18] works.

The timing diagram in Figure 4.7 shows the small hashes bouncing back and forth between A and B. For the purposes of this example, imagine that the IPID field is split in two and the upper byte is used for A's hashes with the lower byte used for B's hashes. The host sending a packet includes a mini hash of its fields in its half of the IPID. Also, for all but the very first packet, the other end host's hash can be echoed back to them. This gives both ends of a connection visibility into modifications in each direction.

Upon receipt of a packet, the receiving end host can perform two checks:

- Does the echoed hash equal what was sent?
- Does the hash from the other end equal a hash of the fields?

The results of these two checks give the host insight over modifications to packets sent and received, respectively.

Due to the short length of the hashes, there is a larger chance of having collisions. This means that there is a sizable chance our check may not work, even with M being nice and not altering fields. We can tolerate this, though, because each packet is another trial and if M is modifying fields (but not necessarily trying to fool us), we are bound to see it with the majority of the packets.

Note that we make a small adjustment to the notation from Section 4.3.1. We substitute h for ${}_Ah$ and g for ${}_Bh$.

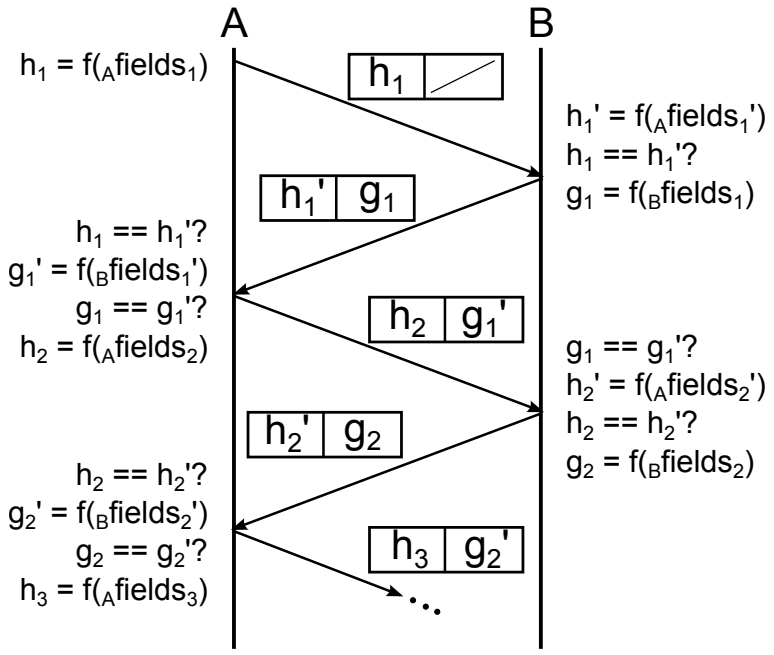

Figure 4.7: Timing diagram with no modifications

4.6.2 Faking Integrity

In order for *M* to fool *A* and *B* into thinking that no modifications were made, it must replace each of the mini hashes exchanged between them. For each packet that passes through *M*, it must recalculate a hash and store the original for up to half an RTT.

4.6.3 Pros

- Simple, easy to understand
- Lightweight, relies on probabilities over multiple trials
- Hash is echoed back to you as the other end saw it

4.6.4 Cons

- Need to use on lengthier connections
- Results may be fuzzy
- Easy for *M* to intercede

4.6.5 Thoughts and Status

This variant seems like it would work well, until middleboxes catch on. Once they do, it would be very easy for a middlebox to redo the hashes. *M* has to do about half of the sum of the work *A* and *B* must do in order to fool them. Not great at raising the bar.

4.7 Variant 4: Hash Striping with Resets

This is an improved variant that transmits the integrity hashes in triplicate in order to withstand a modification to one of the integrity fields. Having 3-way striping of the hash gives reasonable proof that HICCUPS is used by the SYN initiator and allows for a TCP RST to be sent when the hashes fail to match.

Throughout Connection:	No, handshake only
Diagnostic Mode:	None
Fields Used:	Initial Sequence Number (ISN) IPID (on first packet) TCP Receive Window (on first packet)
Raises Bar on *M*:	Not really. *M* must recalculate 2 hash values and at most store 1 packet header for up to half of an RTT.

4.7.1 Detailed Description

The impetus for this variant came after performing initial tests on PlanetLab using the standard opportunistic HICCUPS. The results are discussed in more detail in Chapter 7, but about 13% of the nodes we tested experienced some combination of either ISN translation or modification of the IPID field. While it was good that we were able to detect that a packet header modification was taking place, we lose visibility to any other changes made to the packet by *M*. This is because our integrity hash is overwritten and we lose the ability to check the smaller subsets of header fields that do not contain the ISN or IPID fields.

This variant solves the issue by copying the integrity hash into three separate fields of the packet header. In addition to the fields we used before, ISN and IPID, we also use the TCP receive window. It was realized that the value of this field is not important during the three-way handshake and could be repurposed. A random salt value is still required in order to ensure proper randomization of the ISN, so all hashes are set at 16 bits in length and the 16-bit salt is placed in the upper half of the ISN with the hash going into the lower half. The layout within the fields is shown below in Figure 4.8.

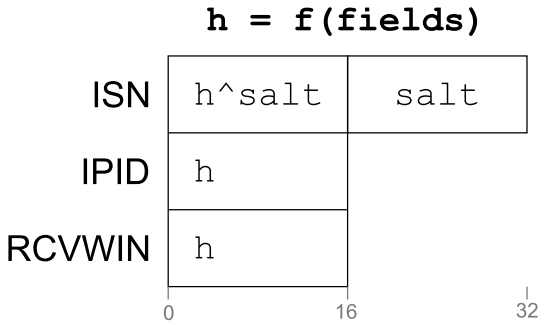

Figure 4.8: Hash and salt layout in header fields

The host that receives the SYN will check to see if any two of the three hash bit ranges match. This will allow transmission of the integrity hash even if one of the three fields were modified by M. A "majority rules" vote is taken from the three fields and that hash is assumed by the receiver to be the HICCUPS hash sent by the SYN initiator. Obviously, if any two of the fields are modified, we lose granularity and can only tell that the path integrity has failed.

A key observation is that it is highly unlikely that any two of the 16-bit fields would be exactly the same unless they were originally set that way by a HICCUPS-enabled SYN initiator. In the worst-case, a randomly set ISN will match either the IPID or receive window value, causing the remote end to infer a HICCUPS capability and calculate path integrity, which will fail. Because of this unlikelihood, we extend this variant with a TCP RST to enable a feedback mechanism.

When the SYN receiver detects a HICCUPS hash (by finding 2 of the 3 hash fields with the same value), and then determines that hash to fail the integrity check, it will respond with a TCP Reset packet. This RST will act as feedback to the SYN initiator that bits were modified while the SYN was in transit to the receiver. It can be differentiated from a RST due to a closed TCP port by sending a SYN without any HICCUPS hashes. In this case, the receiver will not find 2 of 3 fields with the same hash and must not respond with a RST since the SYN initiator is assumed to be not HICCUPS-capable.

If the hashes both exist and pass integrity checks, a similar layout is used to transmit integrity in the SYN-ACK. An example transaction is shown in the timing diagram in Figure 4.9.

All other aspects of this variant are similar to the Opportunistic variant in Section 4.4. Both parties are informed about the path integrity and it fits with the TCP handshake, so only a single RTT is required for integrity status to be obtained. The only downside is that we now use smaller hashes, 16 bits long instead of 32.

4.7.2 Faking Integrity

In order for M to fool A and B into thinking that no modifications were made, it must simply recalculate ${}_{A}h$ and ${}_{B}h$ after performing its modifications. M does not need to regenerate salt values; it can reuse the ones chosen by A and B. Finally, M must be able to store the SYN fields until it can calculate ${}_{B}h$. At most, this will be until it sees the SYN-ACK return from B. This is exactly the same as for the Opportunistic variant, but instead the hash has to be written three times.

One possible weakness of this variant (depending on the viewpoint) is that, along with the receiver, any middleboxes along the path of the packet can tell that the SYN initiator is

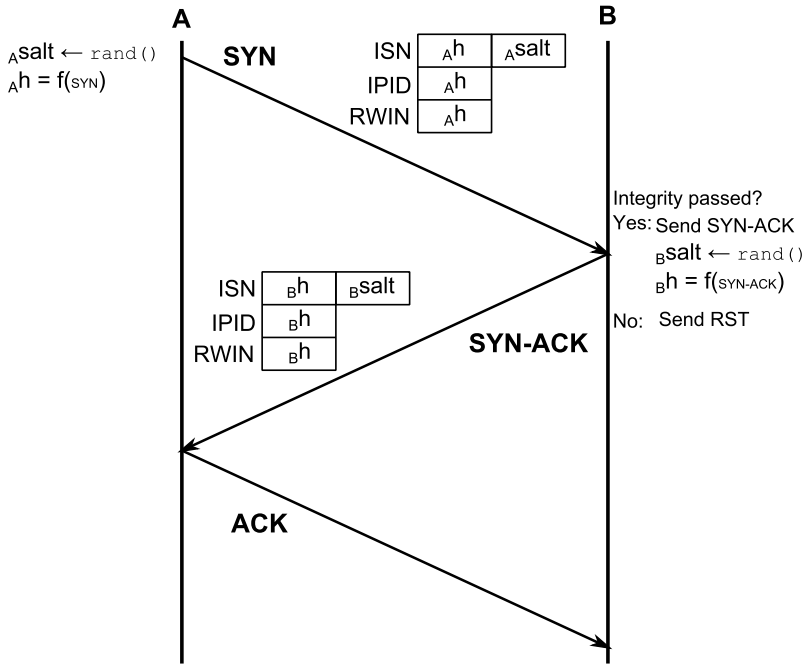

Figure 4.9: Timing diagram with no modifications

using HICCUPS. This saves a devious middlebox from having to overwrite hashes on all SYNs it sees and instead just focus on ones where two out of the three fields have the same value.

4.7.3 Pros

- Interoperable, incrementally deployable
- Withstand modifications to any one of the three fields used to transmit integrity
- Gives status feedback through RST packet (stopping the connection before it starts and allowing the initiator to retry with less features enabled)
- Can distinguish between HICCUPS-capable and a failed integrity check

4.7.4 Cons

- Does not protect the entire connection
- Still breaks if any two integrity transmission fields are modified
- Uses smaller length hashes and would be prone to more collisions

4.7.5 Thoughts and Status

Fairly simple to understand, but implementing the RST work may be difficult in kernel. The variant's key features of feedback and hash modification tolerance are definitely needed

after seeing PlanetLab results. RST can make things messy for non-HICCUPS hosts, however, and if we do not do the RST how else can we transmit the status feedback?

4.8 Variant 5: Hash Rainbow

This variant is similar to the previous variant in Section 4.7, except that the TCP RST is not used as feedback response. Instead, four bits are taken from each hash and used to carry the status. To avoid collisions while allowing for smaller hashes, a different hash function is used for each of the three hashes.

Throughout Connection:	No, handshake only
Diagnostic Mode:	None
Fields Used:	Initial Sequence Number (ISN) IPID (on first packet) TCP Receive Window (on first packet)
Raises Bar on *M*:	Not really. *M* must recalculate 6 hash values and at most store 1 packet header for up to half of an RTT.

4.8.1 Detailed Description

This variant transmits an integrity representation in three places, the ISN, IPID, and TCP receive window. For each of the three fields, the integrity input is hashed using one of three different hashing functions. For example, the hash we place in the ISN may use MD5, while the hash we place in the IPID uses SHA-1 and the hash in the receive window uses SHA-256. The layout is described in Figure 4.10.

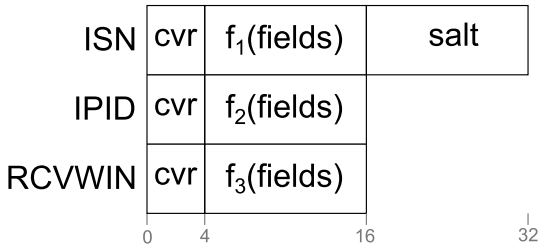

Figure 4.10: Hash and salt layout in header fields

The reason behind this "rainbow" of hashes is that the hash values themselves are only going to be 12 bits long. That means there is a 2^{-12}, or $\frac{1}{4096}$, probability that a random number would be misinterpreted as a valid hash showing correct integrity. Since this probability is fairly high, the multiple hashing functions are used to reduce the chance of a false positive. The chance that the values placed in any two fields by a non-HICCUPS sender would match the expected outputs of two different hashing functions should be much lower.

Since the hashes have been reduced in length to 12 bits, that leaves 4 bits to be used for transmitting status information. In the SYN, these 4 bits carry the coverage type that the SYN initiator would like for the SYN receiver to use when it builds the integrity in the

SYN-ACK. On the returning SYN-ACK, the 4 bits carry the status of the SYN integrity. The transaction is summarized in Figure 4.11.

For the status bits on the SYN-ACK, the lowest order bit is used to signify whether the hash in the RCVWIN field of the SYN matched. The next lowest bit signifies a match in the IPID hash, and the third bit signifies a match in the ISN hash. The highest of the four status bits is always set to 1 so that the TCP receive window value will not go lower than 32,000.

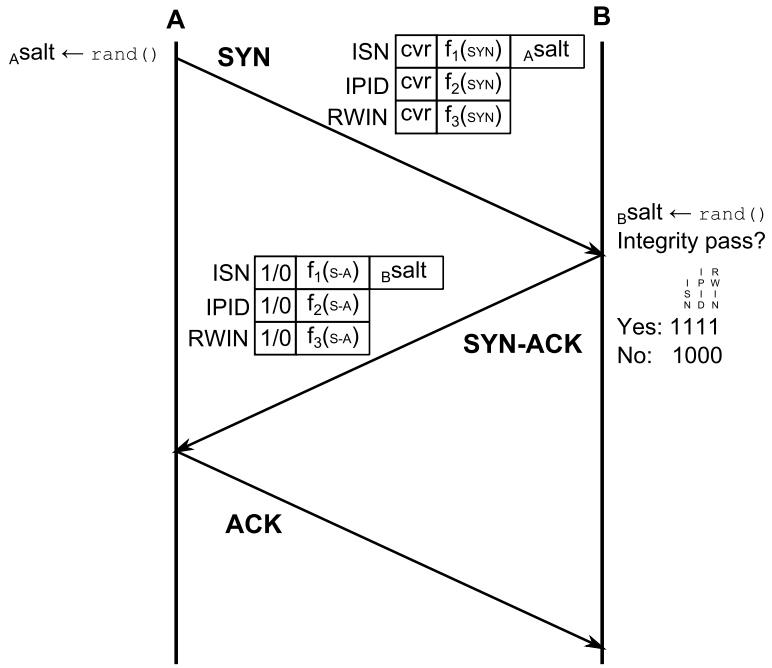

Figure 4.11: Timing diagram with no modifications

4.8.2 Faking Integrity

The actions that need to be taken by a middlebox and the issues involved are the same as with the variant in Section 4.7. The only difference is that, due to the different hash functions, middleboxes can no longer immediately tell that a packet is HICCUPS-enabled. This fact forces a devious middlebox to overwrite hashes on all packets if it wants to fake integrity. The middlebox will also have to perform all three different hashing functions for each packet it modifies.

4.8.3 Pros

- Interoperable, incrementally deployable
- Withstand modifications to any one of the three fields used to transmit integrity
- Gives status feedback

- Won't disrupt any connection attempts due to RST

4.8.4 Cons

- Does not protect the entire connection
- Still breaks if any two integrity transmission fields are modified
- Uses smaller length hashes and would be prone to more collisions (but is helped out by the three different hash functions)
- Can't distinguish between non-HICCUPS capable and failed integrity check (but at least the middlebox can't either)

4.8.5 Thoughts and Status

Seems like the best option. Primary concern with this strategy is the small size of the hashes. We have not quantified how good of a job the rainbow of hash functions does at reducing collisions in the hashes.

But this variant has many good qualities, listed above in the "Pros" section. This combination of good qualities is not present in any of the other variants, making this variant very enticing, but the concern of the small hash sizes must be managed.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Protecting Integrity

The design variants described thus far are effective at detecting packet header manipulations and communicating integrity. Modifications by legacy middleboxes that are either unaware or unable to recognize a connection with HICCUPS integrity will be readily exposed.

However, we must work under the assumption that future middleboxes will have full knowledge of the HICCUPS protocol and could even be engineered to evade detection. A deceitful middlebox could modify the packet headers how it chooses and then rewrite the integrity values used by HICCUPS to detect those changes. In other words, the middlebox could fool each of the two end hosts in the connection into thinking no tampering took place.

Since our design constraints preclude the use of a stronger construction, e.g., a keyed-HMAC, we cannot outright prevent a middlebox from recalculating the integrity. Instead, we strive to add a lighter degree of protection to the integrity and raise the bar on the level of difficulty for a middlebox to make modifications to packets undetected.

5.1 Raising the bar on the middlebox

In order to make it more difficult for a middlebox to recalculate hashes and cover up its modifications, we need to include a secret into that hash that only the endpoints know. However, with no out-of-band channel between the two end hosts and no Diffie-Hellman-like setup, coming up with such a shared secret is difficult. Instead, we look for things that would be difficult for a middlebox to know, but much easier for one or both of the endpoints to know.

As long as one of the endpoints in a connection has such a secret, the integrity value can be encoded with that secret and a middlebox will not be able to replace it with another valid value. Although a true shared secret may not exist, we can still protect the integrity as long as the secret stays hidden from the middlebox long enough to force it to forward the initial packet. If we reveal the secret after the middlebox has already forwarded the packet for us, it will no longer be able to change the integrity and the other end host can decode the integrity value. These "ephemeral secrets" could be any property of a connection that is known only to the sender at the start of the connection. Some possibilities include:

- When a conversation might start
- The parties involved (IPs, ports, etc.)
- Length of a connection (time, bytes, number of packets)
- Timing of individual packets

- Residual TTL
- Proof-of-work
- Application data

There could also be something random that is decided by one of the endpoints, like a coin flip. The overall goal is to add protection to the integrity while imposing as little of the increased burden as possible on the end hosts. The sending host only has to encode the integrity value and the receiving end host only has to store the received integrity until the secret is revealed.

The rest of the sections of this chapter describe design variants that utilize this technique to add protection to the integrity value during its transmission.

5.2 Variant 6: CoinFlips

In this variant, a coin flip is added to the probabilistic variant from Section 4.6 to try to raise the bar on *M*.

Throughout Connection:	Yes
Diagnostic Mode:	None
Fields Used:	IPID
Raises Bar on *M*:	Yes. *M* must do at least as much work as either endpoint. Each RTT it must calculate 5 hashes and store 3 for up to half an RTT.

5.2.1 Detailed Description

At its core, this variant is the same as the probabilistic variant described in Section 4.6. Except now, we have added a bit of randomness to how each side encodes the hash it echoes. This forces *M* to do some calculations to determine the result of the coin flip so that it knows how to encode the echo hash on the return packet.

In the conversation shown in Figure 5.1, *A* initiates the active open and *B* handles the coin flips. It is *A*'s job to determine the value of the flip and use it to properly encode the hash it is about to echo back to *B*. *B* then checks to ensure that the echoed hash was encoded with the same side of the coin that it used.

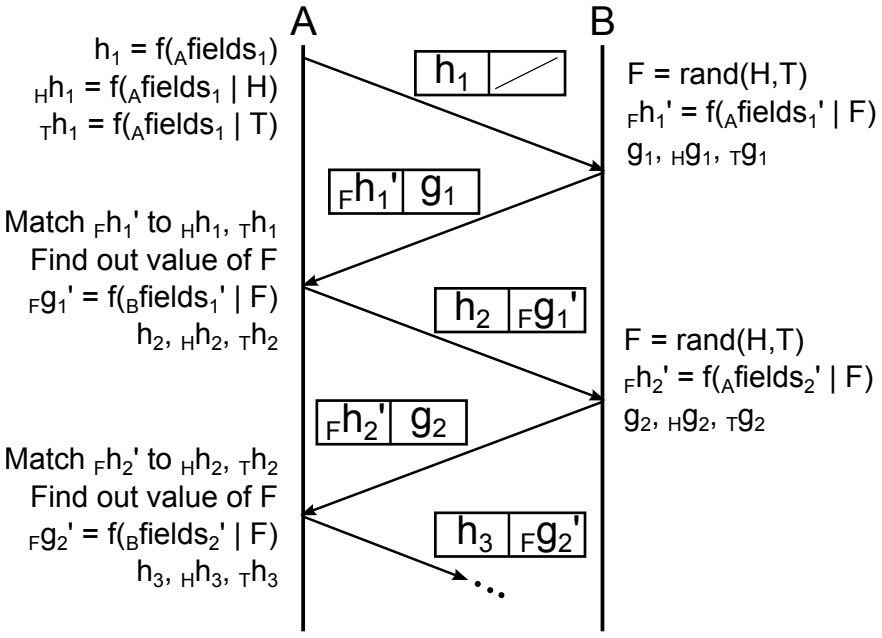

Figure 5.1: Timing diagram with no modifications

For the notation, we make the same adjustment as the last section where we substitute h for ${}_{A}h$ and g for ${}_{B}h$. We use the pre-subscript to denote the value of the coin flip.

5.2.2 Faking Integrity

This variant includes lots of extra steps over the Passing Hashes variant, but it does raise the bar more on M. In order to force more work upon M, we must do a little more ourselves as well. The question is: did we make M do more extra work than we had to do? The conversation where M tries to fool is shown in Figure 5.2.

In order for M to fool A and B into thinking that no modifications were made, it must calculate 5 hashes per RTT and store 3 for up to half an RTT. This is a much greater burden on M than the calculate 1, store 1 requirements of the Passing Hashes variant.

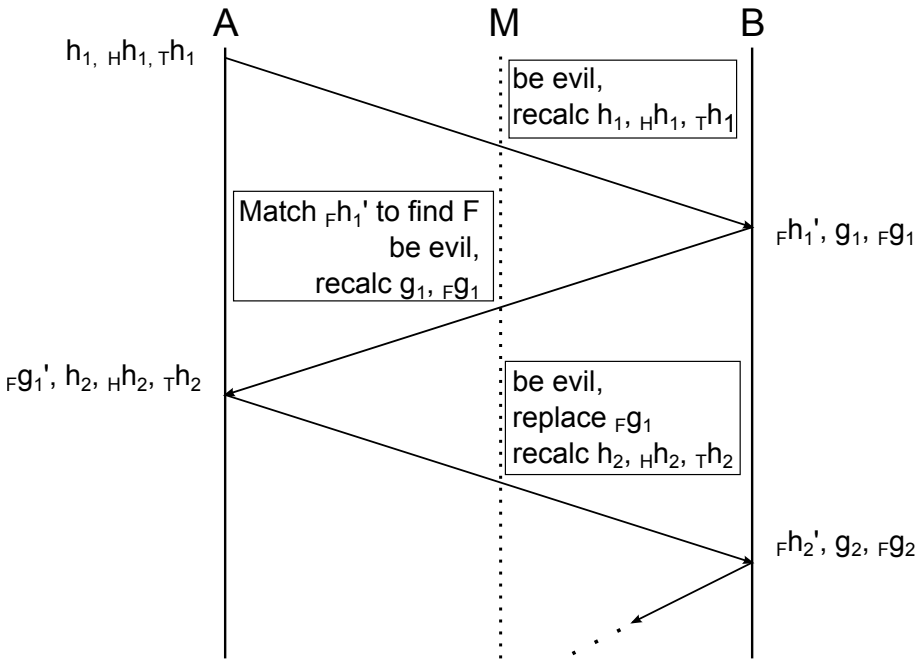

Figure 5.2: Necessary actions to fool A and B

However, we have also increased the load on A and B. Per RTT, A and B must calculate up to 4 hashes each and store up to 2 hashes. The good thing though, is that we have raised the bar on M to just above the work required by either A or B. So in order for M to fool us, it must work harder than either end point.

5.2.3 Pros

- Raised the bar a bit on M, but not up to the sum of the work of A and B
- Has many of the same pros of the probabilistic variant

5.2.4 Cons

- More expensive than the probabilistic variant
- Still needs many trials (packets) to get a good reading

5.2.5 Thoughts and Status

Shows much promise, but does not quite get us all the way where we wanted to be: where M has to do as much work as A and B combined. Still good though. We tested some other closely-related variants, but not seemed to get the ratio of M's work to A's work as high as CoinFlips did.

5.3 Variant 7: HashCash

The goal of this variant is to stop a middlebox from easily overwriting fields by requiring the hashes to have a specific property.

Throughout Connection:	No, handshake only
Diagnostic Mode:	Yes, diagnostic only *Mode Hidden?* No, would be mostly detectable
Fields Used:	Initial Sequence Number (ISN) IPID
Raises Bar on *M*:	Yes. After modifying a packet, *M* must spend CPU cycles to find a good value of R if it intends to fool *A* and *B*.

5.3.1 Detailed Description

In this variant, we require the hashes to show that some computation work was accomplished by the originator. One such way we could do this is to require that the hashes end in a minimum number of zeros. In order to generate a hash that has this property, a system must essentially brute force different values to include with the input to reach the desired property on the output. In our notation, we will call this special value *R*. When a valid *R* is given as input to the hashing function along with the field state representation, it should produce an output with at least the required number of zeros at the end.

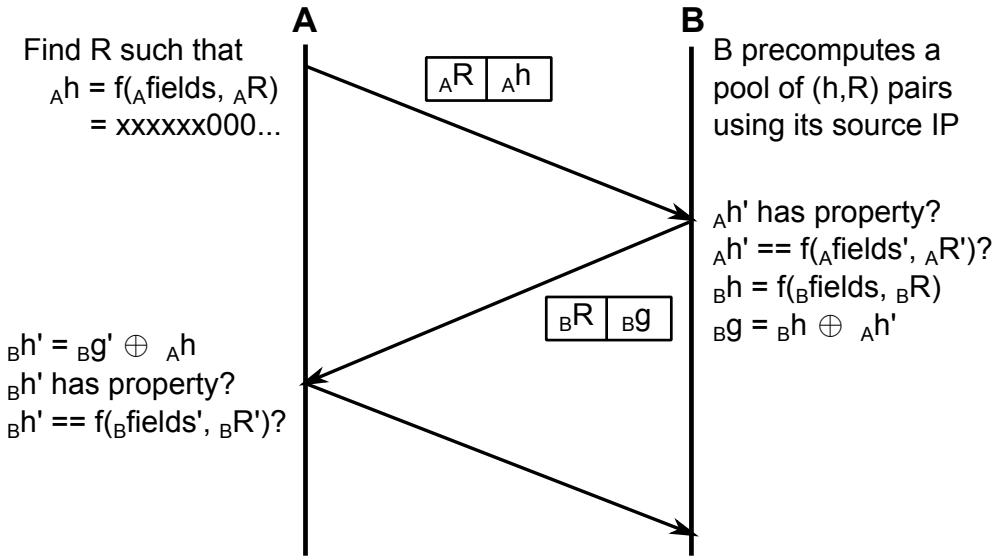

Figure 5.3: Timing diagram with no modifications

We then leverage the fact that *M* does not know when a given connection will start, nor what parameters it will have. This means that *M* cannot start working on the puzzle until

it sees the SYN packet come through. The connection initiator, A, took some time to solve the puzzle before it sent the SYN. This means that there was some lag in starting the connection, but this is less of an issue if it is a diagnostic connection.

After A begins the connection by transmitting the (h,R) pair, B performs two checks on the pair:

1. Does h have the property (end with enough zeros)?
2. Does h equal a rehash of the packet's fields?

B then responds with an (h,R) pair of its own that it has precomputed with some common values and its source IP. We will have to specially craft the set of fields for this check so that B can perform this as precomputation.

5.3.2 Faking Integrity

In order for M to fool A and B into thinking that no modifications were made, it must quickly make its changes and calculate a new (h,R) pair as soon as it sees A initiate the active open. Depending on how difficult we tweak the puzzle, this should take a detectably long enough amount of time. M will have to delay the packet and we can look for the abnormally long RTT.

5.3.3 Pros

- Good at discouraging a middlebox from interfering

5.3.4 Cons

- Forces the endpoints to spend CPU cycles solving hash puzzles
- Lag time from when A's user requests a connection until it solves the puzzle and builds the SYN

5.3.5 Thoughts and Status

This is a big step forward over the previously discussed variants at raising the bar on the middlebox. Ultimately, we believe that the **cons** listed above would force this to be only used in a diagnostic mode connection. This gets into a question of can M tell whether we are in the diagnostic mode or not.

If M can easily detect when two hosts are in diagnostic mode, it can just play nice in those cases and change packets in all the rest of cases. In this variant, there is nothing to disguise the mode. If a middlebox sees hashes that do not satisfy the two checks, it can freely modify packets. Variants in subsequent sections try to tackle this problem.

5.4 Variant 8: Reverse Hash Chain

The salient feature of this variant is that it hides the existence of a check from *M* until the full hash chain is revealed. All of the chain's hashes look random until the salt is revealed.

Throughout Connection:	Yes
Diagnostic Mode:	Optional *Mode Hidden?* Yes, until chain revealed
Fields Used:	IPID
Raises Bar on *M*:	Yes for detection, but *M* can still easily overwrite chain. Can also make a strong argument using a random sampling of checks.

5.4.1 Detailed Description

This variant employs a reverse hash chain to obscure whether the hashes are a check or just random bits. Instead of directly embedding the hashes in the packet, we run it through the hashing function several more times and embed the final output. This results in a chain of values where it is really easy for a computer to determine a relationship in one direction, but not the other. By starting off the connection with the end of a hash chain, we make it very difficult for a middlebox (and the other endpoint) to trace the chain in reverse and determine the original value.

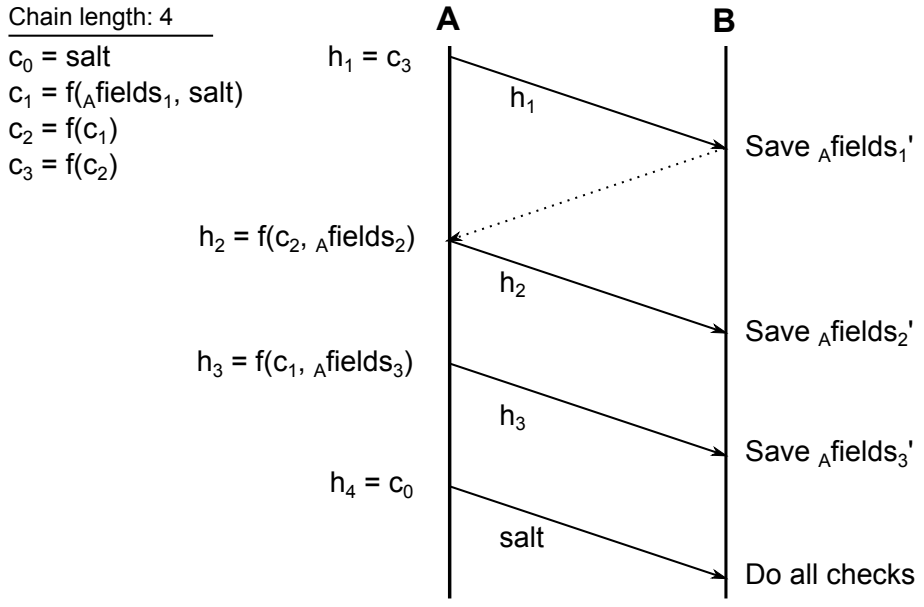

Figure 5.4: Timing diagram with no modifications

Figure 5.4 shows an example conversation using the reverse hash chain check. For illustration purposes, we fix the length of the chain at **four packets**, but it can be any preset

length. The length can be tuned according to how long you want to delay detection. The final (in our case, fourth) packet of the chain reveals the information needed to reconstruct the chain. We call this random value the salt. The salt is needed because it prevents *M* from reconstructing the chain on the first packet.

The following order of events occurs for the length 4 chain:

1. *A* chooses a random salt value, c_0
2. *A* hashes (c_0, $fields_1$) to get c_1
3. *A* hashes c_1 to get c_2
4. *A* hashes c_2 to get c_3
5. *A* embeds c_3 into first packet and sends it
6. *A* embeds the hash of (c_2, $fields_2$) into the second packet
7. *A* embeds the hash of (c_1, $fields_3$) into the third packet
8. *A* embeds the salt, c_0, into the fourth and final packet

Now *M* and *B* can both reconstruct the chain and verify the fields hashes. The extra hashes in steps 5 and 6 provide integrity over the middle packets of the chain. *B* can go back and check these too once it gets the salt.

The key effect we have had is that *M did not know until the end of the chain whether we were actually doing a check*. On the first packet, we force *M* to commit to either:

- overwriting the chain (which it can easily do), or
- leaving the hashes unmodified

If *M* always chooses to overwrite just to be safe, it will be doing more work than necessary since some connections will not use a check. Therefore, the burden on *M* is much greater than on the endpoints, since they only have to expend the hash chain computations when they decide to do a check. If *M* fails to overwrite the chain beginning with the first packet, the connection will fail our checks and we can detect it.

5.4.2 Faking Integrity

As mentioned before, it is easy for *M* to overwrite the hashes and replace them with its own. It only needs to do the recalculations. What this variant makes difficult is detecting the check until the end of the connection, so we will discuss that here.

Suppose *M* sees the packet and wants to tell if a check is being used. Examination of the field holding the hash looks like random bits. The only other option is to try to reconstruct the chain. To do this, two things are needed:

1. the fields over which the hash chain is based

2. the salt

With the first packet, M has the first item. But it still needs the salt, which is not disclosed until the last packet in the chain.

From here, M's only option is to try to hang on to a group of packets in an attempt to capture the salt from the last packet before it has to forward along the first packet. Depending on how long the chain is and how the connection is being used, this can deadlock or wedge the connection because responses (flow control updates, application messages, etc.) from B may be needed to elicit the rest of the chain from A.

5.4.3 Pros

- M cannot detect if a connection is HICCUPS-enabled until the end of the chain
- Can make an over-zealous M do more computation than you, thus raising the bar

5.4.4 Cons

- M can blast over the chain and insert its own, faking integrity
- Endpoint do not get integrity feedback until the end of the chain

5.4.5 Thoughts and Status

This variant stands apart from the rest in its ability to prevent detection of the check until the reveal is done.

We make the "raising the bar" argument by employing randomness in our protection strategy. The idea here is that we randomly protect some $1/N$ connections or $1/N$ packets. Since the middlebox cannot easily guess which packets are protected, he must overwrite hashes on all N of them if he wants a guarantee to fool us. This can be detected when we start seeing valid hash chains for connections and packets which were never protected in the first place.

Furthermore, since our solution is incrementally deployable, there may be connections that never run a check, and M will have to sort through those as well (although it could keep a history of hosts that never embed valid chains, but this is still extra work).

As we will show in the next two sections, this technique can also be combined with HashCash and AppSalt to make them stronger and further raise the bar on M.

5.5 Variant 9: HashCash with Reverse Hash Chain

This variant is a combination of HashCash and reverse hash chains. It requires the original hash of the chain be a special HashCash hash.

Throughout Connection:	Yes
Diagnostic Mode:	Yes, diagnostic only
	Mode Hidden? Yes, until chain revealed
Fields Used:	IPID
Raises Bar on M:	Yes, stronger than HashCash and reverse hash chains.

5.5.1 Detailed Description

Similar to the HashCash variant, A and B can precompute a pair (h,R) where R is a value that is added to the input of the hashing function. The value R causes the output h to have a property which is easily checked. Such a property could be that the hash begins with a minimum number of zeros.

In this variant, we now use the reverse hash cash to obscure the HashCash hash within a chain. The setup is the same as described for the reverse hash chain variant. The only differences are that the R value is used as the salt given by the `REVEAL`, and the original hash of the chain has the HashCash property.

The variant is outlined in Figure 5.5. As with HashCash, A brute forces through different salts to find a resulting h that meets the specified property. More specifically, the resultant hash h must be an element of the set of all hashes that meet the property, or $h \in P$.

The following order of events occurs for a length 4 chain:

1. A brute force searches for a value that yields h such that $h \in P$. This value is c_0.
2. A hashes (c_0, $fields_1$) to get c_1
3. A hashes c_1 to get c_2
4. A hashes c_2 to get c_3
5. A embeds c_3 into first packet and sends it
6. A embeds the hash of (c_2, $fields_2$) into the second packet
7. A embeds the hash of (c_1, $fields_3$) into the third packet
8. A embeds c_0 into the fourth and final packet

Basically it is exactly the same as the reverse hash chain variant, but we replace the random salt value with one that requires computational cycles. This makes it difficult for any middlebox to overwrite the chain since it must start with a value that is valid for the HashCash scheme.

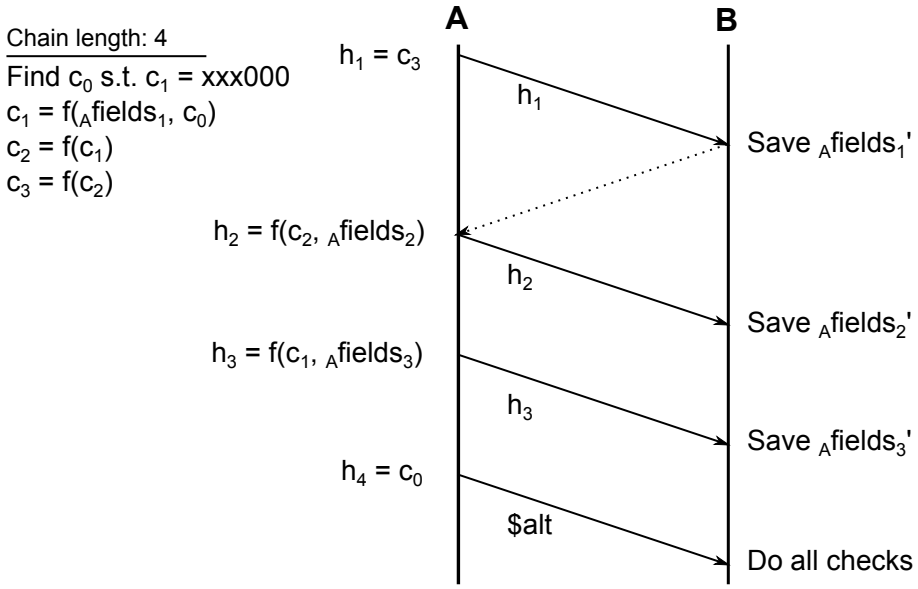

Figure 5.5: Timing diagram with no modifications

5.5.2 Faking Integrity

In order for *M* to fool *A* and *B* into thinking that packets were not modified, it would have to overwrite the entire hash chain while ensuring that the HashCash property still holds. All by the time the full chain is sent, which should be very difficult. Also, if *M* adds a bunch of delay to the final packet of a chain, we should be able to detect that.

5.5.3 Pros

- A good combination of the strengths of HashCash (raising the bar computationally) and reverse hash chains (making mode detection difficult)

5.5.4 Cons

- Could only be used in a diagnostic connection

5.5.5 Thoughts and Status

This variant is worth continuing to explore. It gives all the same benefits of the reverse hash chain method with the added computational burden from the HashCashes. It can basically be viewed as an add-on for the reverse hash chains. Perhaps a stronger assurance mode for tougher middleboxes.

Coding the HashCash pool will be difficult if a CPU must precompute them in spare cycles. Without precomputation, it will probably be unusable for anything other than a diagnostic connection.

5.6 Variant 10: AppSalt

Uses application data in the integrity hashes to make them hard to modify without affecting the user experience.

Throughout Connection:	Yes
Diagnostic Mode:	No
Fields Used:	ISN, IPID
Raises Bar on *M*:	Yes. *M* would have to be a terminating proxy and cache lots of packets.

5.6.1 Detailed Description

This variant builds on the opportunistic approach from Section 4.4 and protects the SYN integrity value with future *application-layer content* from a data packet *yet to be sent*. This ephemeral secret is difficult for a middlebox to reliably determine *a priori*. As before, the integrity value is encoded in the ISN of the SYN, but now the receiving end host, 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, the first data packet need not be a full MSS, e.g., in the case of an HTTP GET request. 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.

To illustrate the complete HICCUPS operation, we present a scenario where a web client connects to a server by performing the 3WHS and then issues an HTTP GET request for a specific resource. Neither the remote server nor any in-path middleboxes can reliably ascertain what will be the application data at the time the SYN is observed. Only the web 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 5.6.

5.6.2 Faking Integrity

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 check whether a connection is HICCUPS-enabled. Encoding integrity with future application data also increases the difficulty for a middlebox to tamper with a packet and evade detection. *M* cannot simply recalculate a new valid integrity. The ephemeral secret forces *M* to process the SYN packet before it can observe the application data. Otherwise, *M* has two remaining options to modify the packet headers and evade detection: make a best guess of the application data,

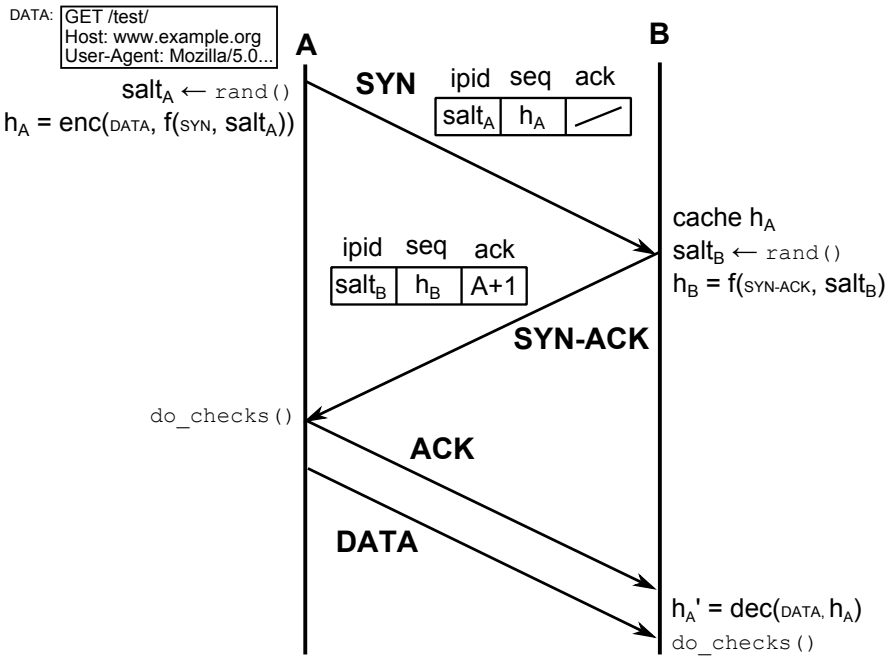

Figure 5.6: Timing diagram with no modifications

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 change the actual application data 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 [4]. 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* would have to be intercepted and the cached data from *A* could be sent. This situation is clearly more complicated than just the translating of sequence numbers; the middlebox has broken a connection and now has to marshal data between them, in addition to sending spoofed packets, buffering data, and rebuilding integrity val-

ues. Further, the middlebox must do this for all connections, potentially representing many endpoints.

5.6.3 Pros

- Very strong against middleboxes intending to perform undetected tampering
- Ties attempted evasion by a middlebox to the user experience

5.6.4 Cons

- Further blurs lines between layers. Forces us to understand application data at the TCP layer.
- Need to modify many applications to provide this data to the TCP stack at connection time

5.6.5 Thoughts and Status

We took the step of verifying that the system call to `connect()` initiates the 3WHS. The SYN is sent before any calls to `send()` are ever made.

Our current implementation of HICCUPS uses this variant as its protection scheme. We had to modify the kernel's socket API so that an application could specially request protection via AppSalt if it desired it. For more details, see Section 6.3.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6: Implementation

We have implemented an integrity transmission mechanism along with an optional integrity protection scheme in a kernel patch for Linux version 3.9.4 [56]. For transmission of integrity, we utilize design elements from the Opportunistic HICCUPS concept described in Section 4.4 and the Hash Rainbow concept described in Section 4.8. To add the optional layer of protection for the integrity transmission, we also implement the AppSalt concept described in Section 5.6.

6.1 Implementation overview

Our implementation of these design elements, which we collectively refer to as HICCUPS, leverages the TCP/IP stack of the Linux kernel to augment outgoing packets and perform special processing of incoming packets to perform the necessary integrity checks. Once a kernel has had our patch applied, it can perform HICCUPS integrity checks with other HICCUPS-capable hosts around the Internet and process the results within TCP.

In order to give an idea of the complexity and breadth of our kernel patch implementation, we calculated the number of lines of code used and found that it took just over 700 lines to implement the core integrity transmission and protection protocols. Table 6.1 shows the complete breakdown of the lines in our patch by their associated functionality.

	Blank lines	**Comments**	**Code**
Core HICCUPS	111	245	560
Debugging	52	72	299
Faking options	4	6	43
AppSalt	33	41	164
Murmur3	51	92	151
Total	251	456	1217

Table 6.1: Lines of code broken down by functionality

Core HICCUPS refers to the integrity transmission and validation components. AppSalt is the protection mechanism. In our implementation, we chose to leverage both the CRC32 and Murmur3 [57] hashes within HICCUPS. Since Murmur3 was not already in Linux, we had to add it, which took about 150 lines of code. The options faking code is only used by our evaluation measurements to imitate the Multipath TCP `MPCAPABLE` [13] response in SYN-ACK packets.

6.2 HICCUPS Details

In order to implement HICCUPS in the kernel, we add hooks at key places where SYN and SYN-ACK packets are processed on both the incoming and outgoing paths. Table 6.2 lists our HICCUPS functions and hook placements corresponding to each type of SYN or SYN-ACK related event. Locations for the hooks were chosen through a combination of code analysis, debugging, and trial and error. The hook locations describe files under the `net/ipv4` source tree within the kernel and the line numbers represent the locations within each file before any editing was done.

Event	HICCUPS function	Hook location
SYN sent	`tcp_hiccups_syn_out`	`ip_output.c:403`
	`tcp_hiccups_after_syn_out`	`tcp_output.c:3029`
SYN received	`tcp_hiccups_syn_in`	`tcp_ipv4.c:1515`
SYN-ACK sent	`tcp_hiccups_synack_out`	`ip_output.c:161`
	`tcp_hiccups_after_synack_out`	`tcp_ipv4.c:863,1652`
SYN-ACK received	`tcp_hiccups_synack_in`	`tcp_input.c:5724`

Table 6.2: HICCUPS functions and hooks, by TCP event

A major challenge working within the kernel is that sequence numbers and IPID values are selected before much of the final packet is built. In particular, since the TCP initial sequence number does not originally depend on the full packet or any information at the IP layer, it can be calculated early on. However, with HICCUPS, we change this design and make the sequence number a function of other fields in the packet header. Therefore, we must postpone calculation of the ISN until the full packet has been created by the kernel and we know the values of all fields we wish to cover. This forces us to go back into the socket structures and update the originally stored sequence number with our new one calculated by HICCUPS.

The two "after" hooks listed in Table 6.2 are also due to the challenge of the kernel needing and using the sequence number before we have the full packet available for hashing. At the point in the code flow which the full packet is available, we are often working with clones of the socket buffer and need to wait until we come back up the transport layer to update the rest of the sequence number fields.

6.3 Appsalt

AppSalt is an optional layer of protection that requires the kernel know the initial byte range of data that an application wishes to send in a connection before the TCP 3-way handshake is initiated. This situation is incompatible with the traditional ordering of socket calls as shown in Listing 6.1. In that sequence of calls, the TCP 3-way handshake will initiate on the `connect()` call and the SYN must be sent before the kernel is even presented with any application data.

Listing 6.1: Old socket call order

```
s = socket(...);
connect(s, addr);
send(s, msg);
```

Listing 6.2: New socket call order

```
s = socket(...);
sendto(s, msg,
       MSG_HICCUPS, addr);
```

In order to implement the AppSalt functionality, we made a small modification to the kernel's socket API. Deriving inspiration from the socket API changes made by the TCP Fast Open extension [58], we added a new flag, `MSG_HICCUPS`, that can be used by the `sendto()` call. Listing 6.2 shows the new sequence of calls that would need to be used by an application requesting AppSalt protection.

We believe that this method of implementation has the best combination of deployability and compatibility properties. An alternative method to implementing AppSalt would be to modify the logic behind the `connect()` call directly so that the handshake is not initiated until the first `send()` call. This would have the positive property of automatically engaging AppSalt protection for all applications without having to update them, but concerns about compatibility and decreased acceptance by the community led us to opt for the more gradual approach.

There are upsides to `MSG_HICCUPS` flag approach. For one, since it is the same method used by TCP Fast Open with just a different flag name, it will be trivial for applications already making use of TCP Fast Open to also use AppSalt protection with HICCUPS. The application need only `OR` the `MSG_HICCUPS` flag with the TCP Fast Open flag in its `sendto()` calls. Many applications have already begun upgrading to include use of TCP Fast Open, including the widely used Google Chrome browser [59].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 7:
Evaluation

We first performed a series of experiments to properly vet our implementation and ensure that both the methodology and coding were correct. First, we deployed HICCUPS-enabled hosts to a virtualized testing environment for initial validation purposes. After ensuring that HICCUPS could detect modifications in a controlled environment where he have ground truth, we deployed implementations to a global Internet networking testbed, PlanetLab [60].

7.1 Controlled Environment

Using virtual hosts running inside Virtualbox, we model a situation where two end systems are connected with a third system along the path between them. The layout is shown in Figure 7.1. Each of the two end systems are running the HICCUPS kernel and the third system acts as a transparent middlebox in their communications.

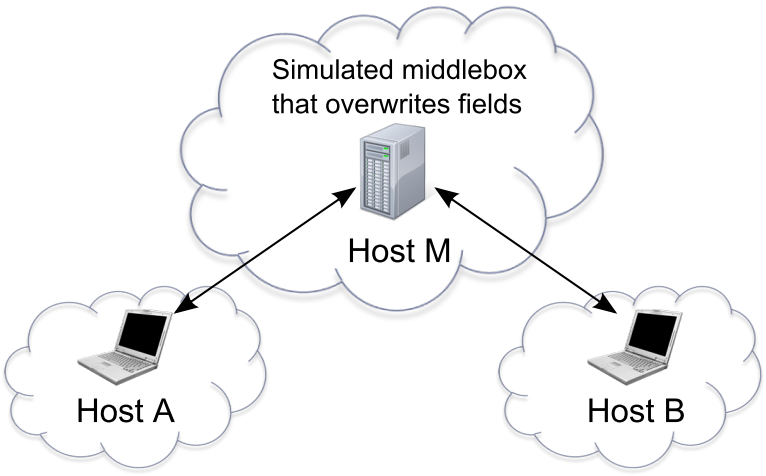

Figure 7.1: Diagram of Virtualbox testing

In order to imitate changes that a middlebox could make, we use an iptables rule to redirect each forwarded packet up to user space via the nfqueue-bindings software [61]. A Scapy [62] script written in Python receives the packet, modifies it, recalculates the network checksum, and then sends it back out on to its destination. We have written the Scapy script so that it can make an array of modifications to packets, all of which are shown in Table 7.1. Each of these is a possible modification we expect a middlebox could make to a passing packet.

Field modified	Description of modification
IP ECN codepoint	If an ECN-capable codepoint is set, zero it out.
	If a congestion-experienced codepoint is set, set one of the bits to zero.
IP DF flag	Complement the Don't Fragment bit
TCP ECN flags	Set both flag bits to zero
TCP ECN flags	Complement each ECN flag bit
IP ID	Set value to zero
TCP MSS	If no MSS is set, set a random one between 1 and 1460 bytes
	If an MSS is set, lower it to a specified value
Reserved fields	Turn reserved bits in TCP and IP headers to 1
TCP Receive Window	Offset the receive window by a specified value
All	All of the above modifications enabled at once

Table 7.1: List of modifications made by middlebox simulator

Using the middlebox script, we tested the effectiveness of the HICCUPS kernel to detect each of the supported modifications. In each case, the two end hosts were able to detect that a change took place when there were in fact changes made, and the hosts did not detect any changes when the Scapy script was disabled.

For the purposes of an example to show how HICCUPS is used, we will more closely examine the scenario of a blocked ECN negotiation. In this scenario, both hosts *A* and *B* of Figure 7.1 are ECN-enabled and request it on their connections. The middlebox script is programmed to set both ECN flags in the TCP header to zero on any packets it sees. As we will see later in Section 7.4.3, this is a fairly common modification on the actual Internet. The modification has the effect of keeping both end hosts from using ECN, even when they both support it.

Working from the point of view of Host *A*, we notice there is a problem when the SYN-ACK that returns from an ECN-capable host has ECN disabled. Running a packet capture during a connection attempt to Host *B*, we see the situation as shown in Figure 7.2. The SYN leaves with the ECE and CWR TCP flags set (to indicate a desire to enable ECN for this connection), but the SYN-ACK returns with neither flag set.

We can confirm that it is a middlebox that is disabling our ECN flags using HICCUPS. Shown in Figure 7.3 is the output from a HICCUPS test client. First, HICCUPS probes are sent from Host *A* to *B* which reveals that a modification is occurring on the SYN, but not the SYN-ACK. Specifically, that modification is to one of the bits covered by the HECN probe type. Second, we disable ECN on Host *A* and try our probes again. Now we see that all of the probes passed the integrity checks, even the ECN probe which previously failed.

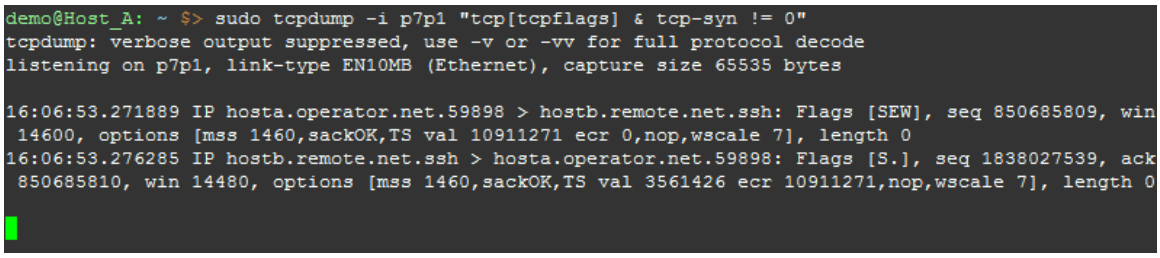

Figure 7.2: Screenshot of packet capture from Host *A*

This is because the middlebox script we have running only changes ones to zeros in the ECN flags. With ECN off, there is no modification to be made.

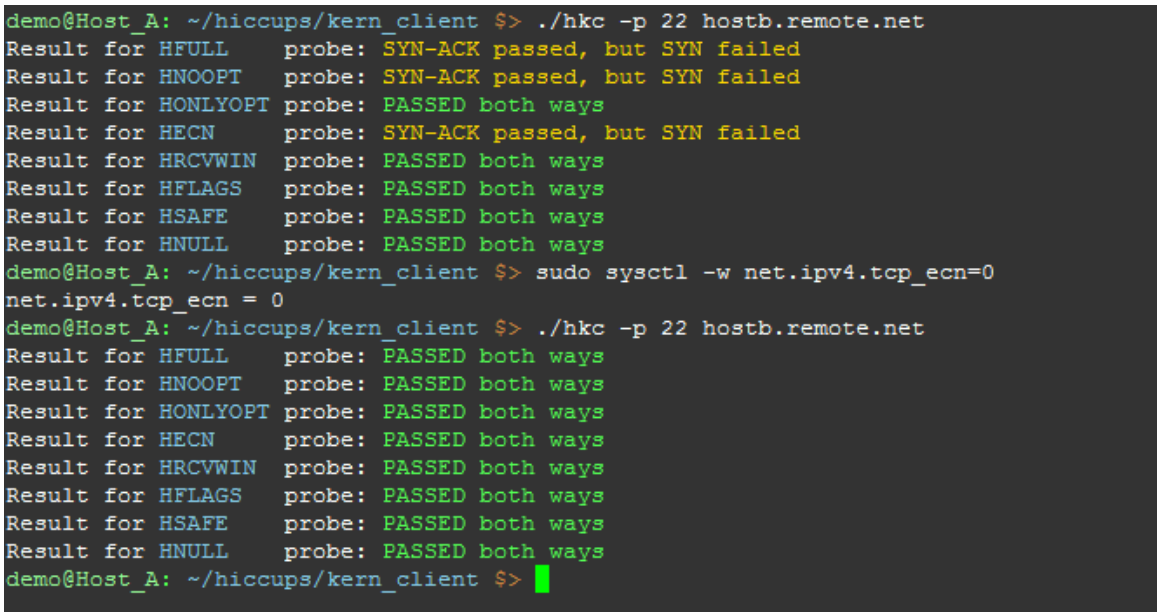

Figure 7.3: Screenshot of HICCUPS test client probing from Host *A* to *B*

7.2 PlanetLab Experimental Description

After successful confirmation that two HICCUPS-enabled hosts could correctly detect packet modifications on their path, we expanded our experimental scope to the Internet. We use PlanetLab [60], a global research overlay network that provides researchers with user-level access to a set of Internet-connected hosts from around the world. Researchers from participating organizations can upload and execute their own programs on the hosts, but cannot modify the kernel. Because of this limitation, we created a userspace raw socket tool that mimics the client functionality of the kernel patch described in Chapter 6.

During the experiment, we tested one node from each unique PlanetLab site, e.g., if UC Berkeley has ten nodes we only test from one of the ten, so as to not be biased towards sites with a large number of nodes. After filtering for these site-unique nodes and leaving out nodes that did not come up, we were able to gain access to 199 different nodes to act as sources.

For each PlanetLab node that we test, we send HICCUPS probes out to three ports (22, 80, and 443) on each of the twelve destination test servers listed in Table 7.2 (except hiccups-icsi, as port 443 was not open inbound).

DNS name	Network	Physical location	Ports
hiccups-mit.cmand.org	MIT	Massachusetts	22, 80, 443
hiccups-ncr.cmand.org	Virginia Tech	Virginia	22, 80, 443
hiccups-icsi.cmand.org	ICSI	California	22, 80
hiccups-mry.cmand.org	Comcast Business	California	22, 80, 443
hiccups-ec2-nva.cmand.org	Amazon EC2	Virginia	22, 80, 443
hiccups-ec2-ore.cmand.org	Amazon EC2	Oregon	22, 80, 443
hiccups-ec2-cal.cmand.org	Amazon EC2	California	22, 80, 443
hiccups-ec2-ire.cmand.org	Amazon EC2	Ireland	22, 80, 443
hiccups-ec2-jp.cmand.org	Amazon EC2	Japan	22, 80, 443
hiccups-ec2-sng.cmand.org	Amazon EC2	Singapore	22, 80, 443
hiccups-ec2-au.cmand.org	Amazon EC2	Australia	22, 80, 443
hiccups-ec2-brz.cmand.org	Amazon EC2	Brazil	22, 80, 443

Table 7.2: PlanetLab nodes connected to each of these servers

From each PlanetLab node to each [server:port] combination, we test all eight probe types a total of five times: with ECN on and off, and with three different values of MSS. The parameters for the five trials are shown in Table 7.3.

	ECN enabled	MSS enabled	MSS value
Trial #1			
Trial #2	X		
Trial #3		X	1460
Trial #4		X	480
Trial #5		X	1600

Table 7.3: Experimental parameters for the five trials

To summarize, there were:

- 199 PlanetLab nodes (sources)
- 35 [server:port] combos (destinations)

  - 12 servers
  - 3 destination ports (minus 443 on hiccups-icsi)
- 8 probe types (HFULL, HECN, etc.)
- 5 probing parameter sets (Table 7.3)

This led to a total of 278,600 probes in the resulting dataset.

7.2.1 PlanetLab node breakdown

In order to further understand the makeup of our set of PlanetLab nodes, we geolocated all 199 nodes using the MaxMind GeoLite country database [63]. The results of that breakdown are shown in Table 7.4 by country and in Table 7.5 by continent. The vast majority (almost 80%) of the nodes reside in either the US or Europe.

# of nodes		Countries
70		United States
13		Germany
11		France
9		Italy
7	ea.	Spain and Poland
6	ea.	Portugal and the UK
5	ea.	Canada, China, Sweden, New Zealand, and Greece
4	ea.	Israel and Brazil
3	ea.	South Korea, Japan, and Switzerland
2	ea.	Turkey, Ireland, Argentina, Belgium, Hong Kong, the Netherlands, Finland, and Hungary
1	ea.	Czech Republic, Norway, Thailand, Ecuador, Australia, Singapore, Jordan, Denmark, Romania, Austria, Taiwan, and Cyprus

Table 7.4: Breakdown of PlanetLab nodes by country

Continent	# of nodes	Percent
Europe	88	44.2%
North America	75	37.7%
Asia	23	11.6%
South America	7	3.5%
Oceania	6	3.0%

Table 7.5: Breakdown of PlanetLab nodes by continent

7.3 Dataset Statistics

We begin by examining the results at a higher level. For the implementation of HICCUPS used during this experiment, there were four possible results from each probe: `PASS`, `NOMATCH`, `ONEWAY`, or `TIMEOUT`. `PASS` means that header integrity was upheld on both the forward and reverse paths between the two hosts. `NOMATCH` means that the SYN-ACK

integrity failed, but tells us nothing of the SYN. `ONEWAY` means that the SYN integrity failed, but the SYN-ACK was upheld. `TIMEOUT` means that no SYN-ACK was received before the timeout expired.

The result type breakdown for the 278,600 probes in the dataset is shown in Table 7.6. The table does not break the results out by trial parameters. The largest amount of non-passing probes occurred for the HFULL probe type, as expected, since it covers all fields. The second most non-passing type was HONLYOPT, which covers only the IP and TCP options. This indicates where the largest proportion of modifications occurred.

	PASS	NOMATCH	ONEWAY	TIMEOUT
HFULL	27769	5822	932	302
HNOOPT	29109	4543	885	288
HONLYOPT	28445	5788	297	295
HECN	29210	4513	795	307
HRCVWIN	29924	4419	175	307
HSAFE	29859	4434	232	300
HFLAGS	29922	4429	175	299
HNULL	29920	4429	175	301
Total	234158	38377	3666	2399

Table 7.6: Probing results, totaled by probe type

7.3.1 Perfect Hosts

There were 59 hosts out of 199 where every single outgoing probe passed HICCUPS checks, or 29.65% of all nodes. We call these perfect hosts since no modifications were made to any of its packets. The sizable number of perfect hosts helps ensure us that there were no middleboxes directly in front of any of our servers. If there were, we would not have any perfect hosts. The distribution of perfect hosts by country is listed in Table 7.7.

# of perfect hosts	Countries
32	United States
3 ea.	Brazil, Poland, Sweden, and New Zealand
2 ea.	Canada, Republic of Korea, Finland, and Japan
1 ea.	Hong Kong, France, Australia, Thailand, Norway, Taiwan, and Denmark

Table 7.7: Perfect host breakdown by country

7.3.2 Timeouts

PlanetLab is a relatively unstable network and each host is shared with many other users. As such, 0.9% of all probes timed out and many of the hosts experienced at least one timeout. 98 of our nodes experienced at least 1 probe timeout at some point, that is 49.25%

of all nodes. However, the vast majority of the timeouts were limited to a small set of the nodes: 4 hosts were responsible for 80.2% of the timeouts.

7.3.3 CDF

Figure 7.4 shows a CDF of the fraction of probes that passed HICCUPS checks for each cumulative fraction of test nodes. The ranges marked as "interesting" and "ISN translation" make up the fraction of nodes that experienced some amount of in-network changes to their packets for at least one of the probes they sent. This range consists of about 70% of all PlanetLab nodes we tested.

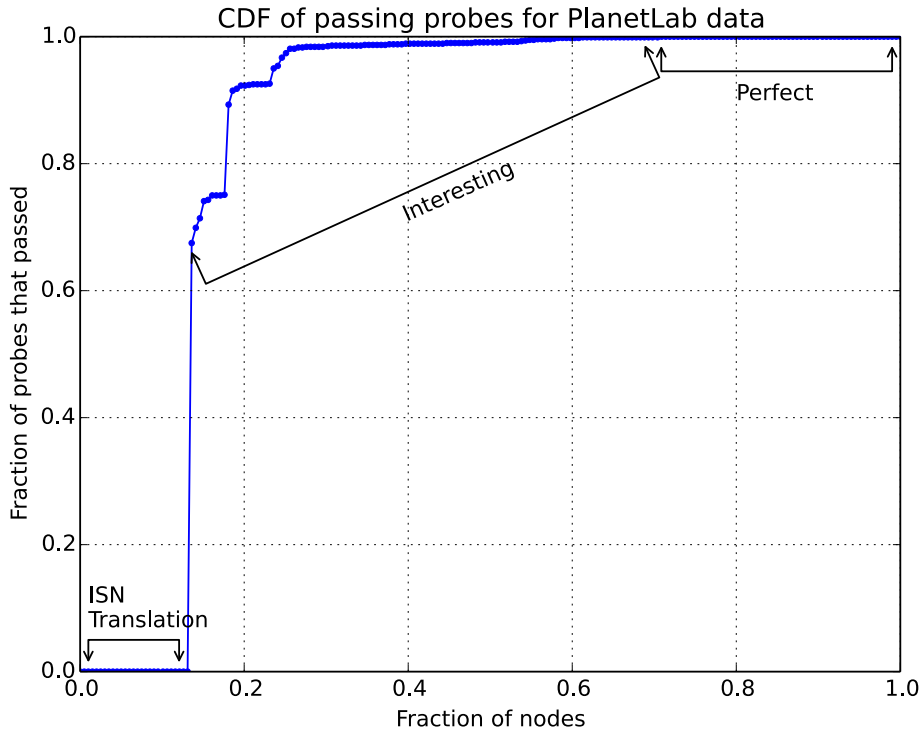

Figure 7.4: Cumulative distribution of passing probes

7.4 Detected Modifications

7.4.1 ISN translation

Translation of the initial sequence number (ISN) will cause the HNULL probe to fail. Since it does not cover any additional fields, it is basically a check of just the integrity carrying fields, the ISN and IPID. About 13.1% of nodes failed HNULL probes, indicating the presence of a middlebox performing translation in front of that node. Here are the exact counts:

ISN or IPID definitely translated:	26
ISN and IPID safe from modifications:	170
Mostly safe, but had path variances:	3

These were the three nodes with path variances: 210.75.225.60, 190.227.163.141, 162.105.205.22. Table 7.8 shows the set of hosts experiencing ISN translation broken down by country.

Hosts w/ ISN translation	Countries
8	United States
2 ea.	France and Sweden
1 ea.	Turkey, UK, Israel, Italy, Portugal, Singapore, Poland, Germany, Spain, Switzerland, Japan, New Zealand, Cyprus, and Ecuador

Table 7.8: ISN translation breakdown by country

7.4.2 TCP Maximum Segment Size (MSS)

Note that in Table 7.9 that there are a large number of NOMATCH probe results no matter the value of MSS that we request. Due to the way our probe coverages are designed, these failed matches may be due to any other modification made to the IP or TCP options. We did not have granularity down to specific options such as MSS in this dataset.

34 hosts never once passed an HONLYOPT probe for any of the MSS values we tried. This is 17.1% of all nodes we tested.

MSS Value	PASS	NOMATCH	ONEWAY	TIMEOUT
None	5676	1163	65	61
480	5712	1146	35	72
1460	5724	1154	35	52
1600	5649	1160	99	57
Total	22761	4623	234	242

Table 7.9: Overall totals for various requested values of MSS

7.4.3 ECN

When we enabled ECN negotiation from the PlanetLab hosts, it resulted in an increase in each non-PASS result category:

	PASS	NOMATCH	ONEWAY	TIMEOUT
HECN with ECN **disabled**	5994	881	34	56
HECN with ECN **enabled**	5267	979	657	62

Note the large jump in the ONEWAY results (SYN integrity failed, but SYN-ACK unmodified). Our inference here is that one or both of the ECE or CWR bits in the TCP header were flipped back to zero. ECN fails to negotiate so the SYN-ACK does not carry the ECN bits and is left alone. There is also an unexplained jump of about 100 in the number of NOMATCH results. Further analysis should be done to determine the cause of these.

Approximately 13 (6.5%) hosts were affected on all paths by enabling ECN:

	ECN off	ECN on
Hosts with majority of HECN probes passing	172	159

Figure 7.5 shows that about 60 more nodes were affected on some fraction of the paths between them and the test servers.

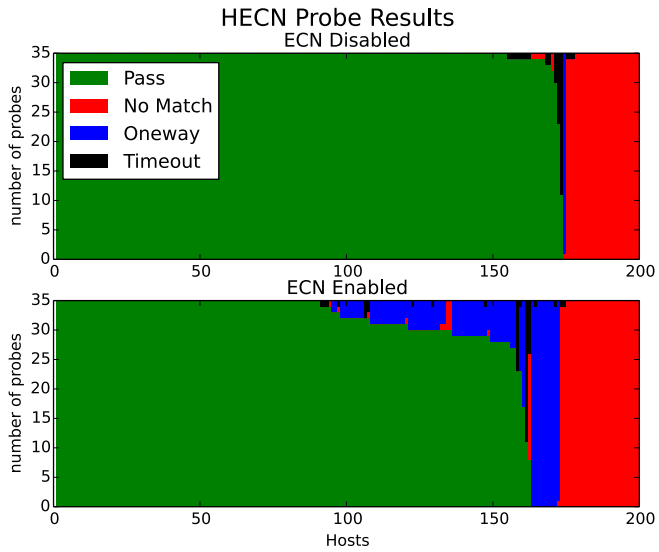

Figure 7.5: Probing results of HECN probe, displayed by host

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 8:
Conclusions and Future Work

Debugging IP network problems end-to-end is a difficult, often manual process exacerbated by the presence of currently opaque middleboxes. In this report, we outline designs for integrity detection mechanisms and integrate those concepts into our implementation, TCP HICCUPS. HICCUPS is a backward-compatible and incrementally deployable extension to TCP that reveals packet header manipulation to both sides of a TCP, thereby facilitating the efforts of endpoints to cooperate with the middleboxes along the potentially asymmetric paths between them.

We evaluate our real-world Linux kernel implementation of HICCUPS on a distributed and diverse set of endpoints and find that HICCUPS discovers a wide variety of (asymmetric) behaviors across thousands of Internet paths. Instances of packet header manipulations we detect include modifications to the TCP sequence number, IPID, ECN, and maximum segment size (MSS). Crucially, packet modification behaviors are discovered by the connection-initiating TCP initiating without other coordination or cooperation with the remote endpoint. We believe that HICCUPS shows the potential to help facilitate the safe deployment of new and experimental options, e.g., ECN and Multipath TCP.

8.1 Future Work

In future work, we wish to utilize the evaluation results to further refine the HICCUPS protocol. In particular, the high presence of sequence number translation disrupted the Opportunistic HICCUPS approach on about 13% of the paths we tested. Spreading the hashes across multiple fields within the header will reduce that value and allow for tests to complete across a greater number of paths. We also wish to devise an efficient search and error correction strategy in order to reduce the number of RTTs required for complete path inference. We plan more extensive performance characterization of selectively enabling ECN and other extensions across real paths whose behavior is inferred by HICCUPS.

We further plan to make our implementations available on our website [64] and invite the community to make use of both the kernel and userspace versions of our solution. More widespread use would help facilitate greater variety in our measurements and further help to refine the protocol. We also hope to utilize comparisons of packet captures and the results from other packet header modification detection utilities such as Tracebox in order to establish ground truths for accuracy evaluation and to do performance comparisons.

We envision several benefits of future widespread HICCUPS deployment beyond improving end-user application performance. In addition to providing invaluable data to re-

searchers and policy makers, implementing HICCUPS in deployed operating systems would enable a new diagnostic capability for network operators. For instance, an operator could determine not only reachability, but also discover any packet modification on both the forward and reverse data plane.

References

[1] H. Schulze and K. Mochalski, "Internet study 2008/2009," ipoque GmbH, Leipzig, Germany, Tech. Rep., 2009.

[2] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and issues," RFC 3234 (Informational), Feb. 2002. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3234.txt

[3] 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 *Proceedings of the 2012 ACM SIGCOMM Conference*, ser. SIGCOMM '12. New York, NY: ACM, Aug. 2012, pp. 13–24.

[4] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: illuminating the edge network," in *Proceedings of the 2010 Internet Measurement Conference*. New York, NY: ACM, 2010, pp. 246–259.

[5] Cisco Systems, "Single TCP flow performance on firewall services module (FWSM)," Oct. 2011. [Online]. Available: https://supportforums.cisco.com/docs/DOC-12668#TCP_Sequence_Number_Randomization_and_SACK

[6] M. Smart, G. R. Malan, and F. Jahanian, "Defeating TCP/IP stack fingerprinting," in *Proceedings of the 9th USENIX Security Symposium*, ser. SSYM '00. Berkeley, CA: USENIX Association, 2000.

[7] G. Fisk, M. Fisk, C. Papadopoulos, and J. Neil, "Eliminating steganography in internet traffic with active wardens," in *Revised Papers from the 5th International Workshop on Information Hiding*. London, UK: Springer-Verlag, 2003, pp. 18–35.

[8] C. B. Smith and S. S. Agaian, "Denoising and the active warden," in *IEEE International Conference on Systems, Man and Cybernetics*, Oct. 2007, pp. 3317–3322.

[9] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics," in *Proceedings of the 10th USENIX Security Symposium*. Berkeley, CA: USENIX Association, 2001.

[10] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, "IP options are not an option," EECS UC Berkeley, Tech. Rep. 2005-24, Dec. 2005.

[11] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend TCP?" in *Proceedings of the 2011 Internet Measurement Conference*. New York, NY: ACM, 2011, pp. 181–194.

[12] S. Bauer, R. Beverly, and A. Berger, "Measuring the state of ECN readiness in servers, clients, and routers," in *Proceedings of the 2011 Internet Measurement Conference*. New York, NY: ACM, 2011, pp. 171–180.

[13] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP extensions for multipath operation with multiple addresses," RFC 6824 (Experimental), Jan. 2013. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6824.txt

[14] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: enabling innovation in middlebox deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ser. HotNets '11, 2011.

[15] A. Medina, M. Allman, and S. Floyd, "Measuring interactions between transport protocols and middleboxes," in *Proceedings of the 2004 Internet Measurement Conference*. New York, NY: ACM, 2004, pp. 336–341.

[16] A. Medina, M. Allman, and S. Floyd, "Measuring the evolution of transport protocols in the internet," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 37–52, Apr. 2005.

[17] K. Ramakrishnan, S. Floyd, and D. Black, "The addition of explicit congestion notification (ECN) to IP," RFC 3168 (Standards Track), Sep. 2001. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3168.txt

[18] N. Spring, D. Wetherall, and D. Ely, "Robust explicit congestion notification (ECN) signaling with nonces," RFC 3540 (Experimental), Jun. 2003. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3540.txt

[19] A. Ramaiah, "TCP option space extension," Mar. 2012. [Online]. Available: http://tools.ietf.org/id/draft-ananth-tcpm-tcpoptext-00.txt

[20] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure, "Are TCP extensions middlebox-proof?" in *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes*, ser. HotMiddlebox '13. New York, NY: ACM, 2013, pp. 37–42.

[21] M. Luckie and B. Stasiewicz, "Measuring path MTU discovery behaviour," in *Proceedings of the 2010 Internet Measurement Conference*, ser. IMC '10. New York, NY: ACM, 2010, pp. 102–108.

[22] Anonymous, "Private communication," 2011.

[23] R. Beverly, A. Berger, Y. Hyun, and k. claffy, "Understanding the efficacy of deployed internet source address validation filtering," in *Proceedings of the 2009 Internet Measurement Conference*. New York, NY: ACM, 2009, pp. 356–369.

[24] S. Kent and K. Seo, “Security architecture for the Internet Protocol,” RFC 4301 (Proposed Standard), Dec. 2005. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4301.txt

[25] S. Kent, “IP authentication header,” RFC 4302 (Proposed Standard), Dec. 2005. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4302.txt

[26] A. Heffernan, “Protection of BGP sessions via the TCP MD5 signature option,” RFC 2385 (Proposed Standard), Aug. 1998. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2385.txt

[27] J. Touch, A. Mankin, and R. Bonica, “The TCP authentication option,” RFC 5925 (Proposed Standard), Jun. 2010. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5925.txt

[28] A. Freier, P. Karlton, and P. Kocher, “The Secure Sockets Layer (SSL) Protocol Version 3.0,” RFC 6101 (Historic), Aug. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6101.txt

[29] A. Langley, “Opportunistic encryption everywhere,” *Web 2.0 Security and Privacy (W2SP)*, 2009.

[30] N. Williams and M. Richardson, “Better-Than-Nothing Security: An Unauthenticated Mode of IPsec,” RFC 5386 (Proposed Standard), Nov. 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5386.txt

[31] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh, “The case for ubiquitous transport-level encryption,” in *Proceedings of the 19th USENIX conference on Security*. Berkeley, CA: USENIX Association, 2010, pp. 26–26.

[32] R. Potharaju and N. Jain, “Demystifying the dark side of the middle: A field study of middlebox failures in datacenters,” in *Proceedings of the 2013 Internet Measurement Conference*, ser. IMC ’13. New York, NY: ACM, 2013, pp. 9–22.

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

[34] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker, “Middleboxes no longer considered harmful.” OSDI, 2004.

[35] IETF, “Transport area working group (tsvwg),” Feb. 2013. [Online]. Available: http://datatracker.ietf.org/wg/tsvwg/

[36] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward software-defined middlebox networking," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, ser. HotNets '12, 2012.

[37] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xomb: Extensible open middleboxes with commodity servers," in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '12. New York, NY: ACM, 2012, pp. 49–60.

[38] G. Gibb, H. Zeng, and N. McKeown, "Outsourcing network functionality," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY: ACM, 2012, pp. 73–78.

[39] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA: USENIX Association, 2012.

[40] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying Middlebox Policy Enforcement Using SDN," in *Proceedings of the 2013 ACM SIGCOMM Conference*, ser. SIGCOMM '13. New York, NY: ACM, Aug. 2013, pp. 27–38.

[41] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[42] "IEEE standard for information technology–telecommunications and information exchange between systems–local and metropolitan area networks–specific requirements part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications," *IEEE Std 802.3-2008*, 2008.

[43] "IEEE standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications," *IEEE Std 802.11-2012*, 2012.

[44] R. Braden, D. Borman, and C. Partridge, "Computing the internet checksum," RFC 1071 (Informational), Sep. 1988. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1071.txt

[45] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 309–319, Aug. 2000.

[46] P. Eckersley, “Switzerland design,” May 2008. [Online]. Available: http://switzerland.svn.sourceforge.net/viewvc/switzerland/trunk/doc/design.pdf

[47] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet, “Revealing middlebox interference with tracebox,” in *Proceedings of the 2013 Internet Measurement Conference*, ser. IMC ’13. New York, NY: ACM, 2013, pp. 1–8.

[48] M. Luckie, Y. Hyun, and B. Huffaker, “Traceroute probe method and forward IP path inference,” in *Proceedings of the 2008 Internet Measurement Conference*, ser. IMC ’08. New York, NY: ACM, 2008, pp. 311–324.

[49] J. Postel, “Internet control message protocol,” RFC 792 (Internet Standard), Sep. 1981. [Online]. Available: http://www.rfc-editor.org/rfc/rfc792.txt

[50] F. Baker, “Requirements for IP version 4 routers,” RFC 1812 (Standards Track), Jun. 1995. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1812.txt

[51] W. Eddy, “TCP SYN flooding attacks and common mitigations,” RFC 4987 (Informational), Aug. 2007. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4987.txt

[52] P. McManus, “Improving syncookies,” Apr. 2008. [Online]. Available: http://lwn.net/Articles/277146/

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

[54] N. R. Bennett, “JPEG steganalysis and TCP/IP steganography,” master’s thesis, University of Rhode Island, 2009.

[55] X. Luo, E. W. W. Chan, and R. K. C. Chang, “CLACK: A network covert channel based on partial acknowledgment encoding,” in *IEEE International Conference on Communications (ICC)*, Jun. 2009, pp. 1–5.

[56] L. Torvalds, “Linux kernel v3.9.4.” [Online]. Available: http://www.kernel.org

[57] A. Appleby, “Murmurhash 3.0,” 2011.

[58] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan, “TCP Fast Open,” in *Proceedings of the Seventh CoNEXT*, ser. CoNEXT ’11, 2011.

[59] Google Inc, “chromium code search,” 2013. [Online]. Available: https://code.google.com/p/chromium/codesearch#chromium/src/net/socket/tcp_client_socket_libevent.cc&sq=package:chromium&type=cs&l=312

[60] 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.*, vol. 33, no. 3, pp. 3–12, Jul. 2003.

[61] P. Chifflier, "nfqueue-bindings." [Online]. Available: https://www.wzdftpd.net/redmine/projects/nfqueue-bindings/wiki/

[62] P. Biondi, "Scapy." [Online]. Available: http://www.secdev.org/projects/scapy/

[63] MaxMind, "Geolite free downloadable databases," 2013. [Online]. Available: http://dev.maxmind.com/geoip/legacy/geolite/

[64] R. Craven, R. Beverly, and M. Allman, "Handshake-based integrity check of critical underlying protocol semantics (hiccups)," 2014, http://www.cmand.org/hiccups/.

Initial Distribution List

1. National Science Foundation
   Arlington, Virginia

2. SPAWAR Systems Center Atlantic
   North Charleston, South Carolina

3. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California