ICT-257422

CHANGE

CHANGE: Enabling Innovation in the Internet Architecture through Flexible Flow-Processing Extensions

Specific Targeted Research Project

FP7 ICT Objective 1.1 The Network of the Future

D2.4: Flow Processing Architecture: Final Specification

Due date of deliverable: 31 December 2011

Actual submission date: September 28, 2012

Start date of project	1 October 2010
Duration	36 months
Lead contractor for this deliverable	University Politehnica of Bucharest
Version	Final, September 28, 2012
Confidentiality status	Public

Abstract

The Internet has grown over the last twenty years to the point where it plays a crucial role in today's society and business. However, its limitations are well-known: the Internet does not provide predictable quality of service, and does not provide a sufficiently robust and secure infrastructure for critical applications. Worse, making changes to the basic Internet infrastructure is costly, time-consuming and often unfeasible: operators are paid to run stable, always available networks which is anathema to deploying new mechanisms.

This document presents the CHANGE architecture which aims to re-enable innovation in the Internet by embracing flow processing as a first class citizen of the network. Flow processing is performed at various points in the Internet called flow processing platforms. These provide the building blocks to allow the Internet to evolve.

Security is a major concern in such an extensible architecture: to avoid unwanted side-effects, flow processing can only be initiated by the owners of the traffic being processed and ownership is proven cryptographically. Further, a simple set of rules restrict what can be done to the traffic for certain changes, reducing the possibility of abuse.

Extensibility is useless unless users understand what happens to their traffic. We propose the concept of network invariants as a novel way to dynamically change the (currently implicit) contract between the users and the network. Users specify what processing should NOT be applied to their packets, and the network checks and enforces such invariants efficiently.

Finally, we discuss a set of practical ways to implement our architecture using existing technologies, as well as how a few scenarios can be deployed using CHANGE .

Target Audience

Experts in the field of computer networking, the European Comission.

Impressum

Full project title	CHANGE: Enabling Innovation in the Internet Architecture through Flexible Flow-Processing Extensions
Title of the workpackage	WP2: Scenarios, Requirements and Arhitecture
Editor	Costin Raiciu, University Politehnica of Bucharest
Project Co-ordinator	Adam Kapovits, Eurescom
Technical Manager	Felipe Huici, NEC
Copyright notice	© 2012 Participants in project CHANGE

Executive Summary

The great virtue of the Internet architecture has been to enable the deployment of an astonishing range of applications and services based on a relatively simple organisation and a set of common abstractions. However, research has shown that the "traditional" Internet architecture model is disappearing. We find through a wide-scale experimental study (presented in chapter 2) what the new Internet looks like: instead of a transparent end-to-end path, there are now segments of transparent IP connectivity interconnected by nodes that perform L4+ processing on flows.

In short, application logic, in the form of middleboxes such as NATs, firewalls, DPIs and a multitude of other applications, now lives in the middle of the network at the behest of network operators. There are two broad consequences to this situation. First and foremost, the lack of transparency makes it difficult to reason about the emergent behaviour of traffic traversing the Internet. The complexity and uncertainty that results from this hinders the deployment of new applications and services; already existing applications such as Skype have to depend on a myriad of techniques just to provide an end-to-end service.

Our recent experience on creating Multipath TCP and standardizing is one of the biggest examples of just how difficult innovation in the Internet has become. The protocol design that was supposed to take a few months has stretched over 4 long years, resulting in a quite complex specification that aims to cover all the possible ways middleboxes could disrupt Multipath TCP traffic; this is despite the fact that MPTCP only uses TCP options to implement its functionality, and that MPTCP is backwards compatible with TCP.

Unchecked, this situation threatens to escalate: innovation will only become harder unless changes are made. Reverting back to the original end-to-end Internet is also not possible; the trend is already entrenched: for all the opaqueness middleboxes induce, they also provide functionality that makes network operators deem them as indispensable.

This report describes the architecture to CHANGE this state of affairs (chapter 3). The architecture embraces the concept of flow processing and the vision of middleboxes but uses it to re-enable end-to-end innovation. The principle is very simple: third parties should be able to instantiate functionality inside the network as long as the functionality provably applies only to the traffic belonging to those parties.

We start by presenting the architecture requirements on a flow processing platform, listing what would be needed to realise a coherent flow processing architecture for today's Internet. We then define the foundations of a flow processing platform, starting from the basic definition of a flow, its labeling and identification.

We decompose the actual flow processing operations into a set of widely applicable primitives, showing both how they may be used to define instances of processing and the security considerations associated with them. To be attractive, a new flow processing architecture must not raise any new security vulnerabilities. For this reason, we describe a set of security rules applied in the CHANGE architecture and explain why they were chosen and what their overall effect is (chapter 4).

Extensibility is useless unless users understand what happens to their traffic. Today, new protocol design (e.g., MPTCP) is very defensive, assuming the worst from the network: we have switched from a very simple contract with the network (deliver my packets unchanged from TCP up) all the way to a situation where any part of the TCP header or payload may be altered. While it is possible to cope with the new contract at the endpoints by simply encrypting everything this is a non-starter because performance will drop significantly. We propose the concept of network invariants as a novel way to dynamically change the (currently implicit) contract between the users and the network. Users specify what processing should NOT be applied to their packets, and the network checks and enforces such invariants efficiently.

We also include motivating scenarios to test both the applicability and flexibility of our approach (chapter 5). This document finishes with a review of related work (chapter 6) and conclusions (chapter 7).

List of Authors

Authors	Felipe Huici (NEC), Olivier Bonaventure (UCL-BE), Laurent Mathy (ULANC), Mark Handley (UCL-UK), Costin Raiciu (PUB)
Participants	NEC, UCL-BE, ULANC, UCL-UK, PUB
Work-package	WP2: Scenarios, Requirements and Architecture
Security	Public (PU)
Nature	R
Version	1.0
Total number of pages	77

Contents

List of Figures

List of Tables

1 Introduction

The basic architecture of the Internet has been embodied for thirty years in a set of protocols and design documents that indicate requirements for Internet hosts and routers. The architecture is a layered one, with diverse layer 2 networks interconnected by IP routers providing an end-to-end datagram service model at layer 3. Above this, at layer 4, sits TCP, providing reliable congestion-controlled in-order delivery of data to applications at the layers above. The concept is simple; routers look at the headers in IP packets, decrement the TTL and forward the packet on to the next hop if they are able, or drop the packet otherwise. TCP then functions end-to-end, with only the end-systems understanding its semantics.

The great virtue of this model is that it is possible to reason about the emergent behaviour of all the concatenated components, at least if they perform according to spec. In particular, new applications and transport protocols can be introduced, and only the end-hosts need to know about them. It is this great generality that has been the Internet's main advantage, allowing a wide range of new applications to be deployed that were unforeseeable when TCP and IP were being designed. It can be argued that the Internet doesn't do any single task terribly well, but it does everything well enough. And in economic terms, 'well enough' is what actually matters. The problem though is that the Internet doesn't really do everything well enough.

Unfortunately the original Internet architecture has not described the real Internet for around fifteen years now. Network Address Translators and firewalls were the first commonly deployed middleboxes that placed layer 4 (or higher) functionality in the middle of the network, not just at the endpoints. More recently it has become common to employ deep-packet inspection to look within packets and perform rate-limiting of "less important" traffic. Transparent web proxies and application accelerators improve performance by employing L4+ knowledge. Intrusion detection systems reassemble flows and snoop everything, as does lawful intercept equipment. And traffic normalizers try to remove ambiguous corner cases that might represent threats or attempts to bypass detection equipment.

There are two key problems that this plethora of enhancements to the original architecture bring. First, they embed the limitations of current protocols and applications within the network, making it very hard to deploy anything new without jumping through hoops to make it look to the network like existing traffic. Second, they make it extremely difficult to reason about what precisely will happen to a packet sent across the Internet, which makes the network fragile and hard to debug.

During our recent work on trying to standardize TCP extensions, it became clear that no-one really knows what will happen to a TCP packet sent across the Internet if it strays from common practice in any way. To create a solid basis for a new Internet architecture we need to understand what the Internet looks like today before we can build the Internet of tomorrow.

In collaboration with Michio Honda from Keio University, we set out to probe the Internet to test its response to a wide variety of possible extended TCP syntax and semantics. Our study, described in detail in Chapter 2, probed 142 access networks in 24 countries (14 of them in the EU) ranging from cellular providers to WiFi hotspots, residential DSL and cable to university networks, plus a number of corporate networks. By carefully crafting TCP segments between a client on each network and our server, we can probe corner case behaviour. As anecdotal evidence indicated that many middleboxes do not treat all ports equally, we repeated the tests using port 80 (HTTP), port 443 (HTTPS) and a port with no special meaning. The results are somewhat eye-opening.

- New TCP options are removed from traffic on 6% of paths. However, on port 80 this rises to 14% of paths.
- TCP sequence numbers are rewritten on 10% of paths. On port 80, 18% of paths rewrite sequence numbers. Of these on port 80, two-thirds implement some form of TCP proxy which splits the end-to-end path into two or more distinct TCP path segments, and acknowledge data before it reaches the final end point. The remainder, and most on other ports, appear to rewrite sequence numbers just to improve initial sequence number randomization.
- 5% of paths will fail if holes are left in the TCP sequence space. On port 80, the number is 15%.
- If an ACK is sent for data that has not yet been sent, 25% of paths will either drop it or "correct" it. For traffic on port 80, this rises to 33% of paths.

Thus, over a third of the paths are keeping state and performing L4+ functionality. This does not include NAT or basic firewall functionality, which is in addition to the figures above but too ubiquitous to be noteworthy these days. Our study also cannot measure rate-limiting triggered by DPI, lawful intercept, or any form of L4+ processing at the server end of the connection on typical Internet paths. We expect that functionality such as intrusion detection systems and server load balancers are commonplace in modern datacenters, but cannot measure it with the methodology used in this study.

It is clear then that the original end-to-end transparent Internet architecture is disappearing, to be replaced by one where there are segments of transparent IP connectivity interconnected by nodes that perform L4+ processing on flows. It is also clear that traffic receives significantly different processing depending on the transport port.

In this context, is it still possible to extend the Internet? Extending IP is not an option anymore [25]; is the same true for TCP? Our experience in the past years on designing Multipath TCP and TCP Crypt to be deployable in the current Internet shows it is still possible to extend TCP, but the extensions must be designed very defensively. For instance, MPTCP includes an additional checksum just to catch application level gateways that alter the TCP payload and change its length. An overview of the design of MPTCP, provided in Deliverable 4.3, shows just how complicated protocol design has become just because of middleboxes. This is by far one of the biggest motivations that has driven us to try and change the current status quo.

It seems futile to attempt to fight this trend of deploying more and more middleboxes - it is too entrenched at this point. IPv6 may help with NATs, but is unlikely to help with all the other reasons network operators install these middleboxes. The only reasonable strategy seems to be to attempt to embrace middleboxes and the concept of flow processing, but to then attempt to provide a unifying architecture in which they have roles that we can reason about and functionality we can communicate with from the end systems. In particular we set out to answer the question *is it possible to provide an architectural framework in which middleboxes could actually enhance our ability to deploy new applications?*

This document presents the architectural framework to be constructed in the CHANGE project. It is at this stage still work-in-progress, and will continue to be revised in the light of further developments of both the platform and the applications that will use the platforms.

We present the overview of our architecture in Chapter 3. The architecture relies on a categorisation of flow processing into a few simple classes that we can reason about, and enforces a few simple security rules that ensure its flexibility cannot be misused. A core part of the architecture is the ability to understand the composition of flow processing functions, which we enable using the novel concept of *network invariants*: these allow endpoints to specify what type of service they expect from the network, and the network can then implement them in the most efficient way possible. In Section 3.7 we discuss some implementation aspects. We reason about its security properties in Chapter 4. We discuss how to implement motivating scenarios in Chapter 5. We review related work in Chapter 6 and conclude in Chapter 7.

2 Understanding the Internet Today

The Internet was designed to be extensible; routers only care about IP headers, not what the packets contain, and protocols such as IP and TCP were designed with options fields that could be used to add additional functionality. The great virtue of the Internet was always that it was *stupid*; it did no task especially well, but it was extremely flexible and general, allowing a proliferation of protocols and applications that the original designers could never have foreseen.

Unfortunately the Internet, as it is deployed, is no longer the Internet as it was designed. IP options have been unusable for twenty years[25] as they cause routers to process packets on their slow path. Above IP, the Internet has benefited (or suffered, depending on your viewpoint) from decades of optimizations and security enhancements. To improve performance [5, 17, 37, 6], reduce security exposure [33, 55], enhance control, and work around address space shortages [42], the Internet has experienced an invasion of middleboxes that *do* care about what the packets contain, and perform processing at layer 4 or higher *within* the network.

The problem now faced by designers of new protocols is that there is no longer a well defined or understood way to extend network functionality, short of implementing everything over HTTP[44]. Recently we have been working on adding both multipath support[26] and native encryption[11] to TCP. The obvious way to do this, in both cases, is to use TCP options. In the case of multipath, we would also like to stripe data across more than one path.

However, it became increasingly clear that no one, not the IETF, not the network operators, and not the OS vendors, knew what will and what will not pass through all the middleboxes as they are currently deployed and configured. Will TCP options pass unchanged? If the sequence space has holes, what happens? If a retransmission has different data than the original, which arrives? Are TCP segments coalesced or split? These and many more questions are crucial to answer if protocol designers are to extend TCP in a deployable way. Or have we already lost the ability to extend TCP, just like we did two decades ago for IP?

In this section we present the results from a measurement study conducted from 142 networks in 24 countries, including cellular, WiFi and wired networks, public and private networks, residential, commercial and academic networks. We actively probe the network to elicit middlebox responses that violate the end-to-end transparency of the original Internet architecture. We focus on TCP, not only because it is by far the most widely used transport protocol, but also because while it is known that many middleboxes modify TCP behavior [16], it is not known how prevalent such middleboxes are, nor precisely what the *emergent behavior* is with TCP extensions that were unforeseen by the middlebox designers.

We make three main contributions. The first is a snapshot of the Internet, as of 2011, in terms of its transparency to extensions to the TCP protocol. We examine the effects of middleboxes on TCP options, sequency numbering, data acknowledgment, retransmission and segmentation.

The second contribution is our measurement methodology and tools that allow us to infer what middleboxes are doing to traffic. Some of these tests are simple and obvious; for example, whether a TCP option arrives or is removed is easy to measure, so long as the raw packet data is monitored at both ends. However, some tests are more subtle; to test if a middlebox coalesces segments it is not sufficient to just send many segments—unless the middlebox has a reason to queue segments it will likely pass them on soon as they arrive, even if it has the capability to coalesce. We need to force it to have the opportunity to coalesce.

2.1 Methodology And Datasets

We use regular end-hosts to actively measure paths in the Internet. Our aim is to test relevant properties that could impact yet-to-be-deployed TCP extensions. We have resorted to active measurement for a number of reasons:

- We need to generate traffic that mimics new TCP extensions.
- We generate artificial traffic patterns such as contiguous small segments or gaps in the sequence space. It is difficult to use passive measurements for this purpose.
- Packets need to be inspected at both sender and receiver for tests detecting TCP option removal, sequence number shifting, re-segmentation, etc.
- We need to test different destination ports including ports not normally in use, as middlebox behavior depends on the destination port.

2.1.1 Testing Tool

Our middlebox inspection tool is called **TCPExposure** and consists of a client and a server tool. The client acts as an initiator of a TCP connection (the end that sends the SYN), and the server acts as a responder. These are a 3000-line program and a 500-line program both written in Python. The initiator and the responder run tests aiming to trigger on-path middlebox actions. The tools send and receive TCP segments in user space via a raw IP socket or using the Pcap library similarly to Sting [47].

The client tool was built to be easy to use, as most of our tests are run by contributors. To maximize reach, the client tool is cross-platform running on Mac OS, Linux and FreeBSD. It is self-contained and only requires Python and libpcap on the host; these come preinstalled on most systems. The client is straightforward to run: all users need to do is to download it, launch a single shell script and post the results.

The responder tool runs on Linux. It does not maintain state for the TCP connections it is emulating; its replies depend solely on the received TCP segments. For example, the responding segment contains SYN/ACK if the responder has received SYN, acknowledges the end of the sequence number, and has the sequence number based on the received acknowledgement (ACK) number. This stateless behavior makes it relatively easy to reason about observed behavior because there is no hidden server state.

2.1.2 Common Procedures

Table 2.1 lists the fixed TCP parameters at the initiator and the responder. These values are used in all our measurements unless stated otherwise.

Table 2.1: Default TCP Parameters

Parameter	**Initiator**	**Responder**
Initial Sequence Num (ISN)	252001	11259375
Window Size	8064	32768
MSS	512	512
Window Scale	-	6
SACKOK	-	1
Timestamp (TS_val)	-	12345678

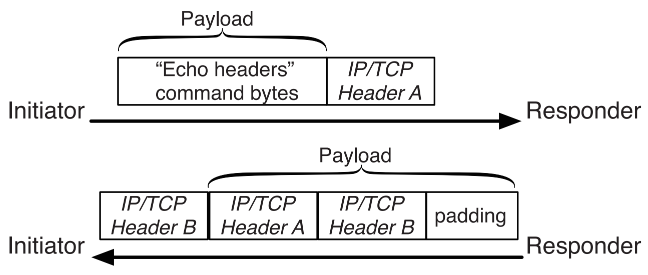


Figure 2.1: Echo Headers Command

We use a 512 byte MSS at both ends, less than what most TCP implementations advertise. This value is smaller than the MTU of most Internet paths, and was chosen to avoid unexpected fragmentation during tests.

We expect middleboxes to behave differently depending on the application type, and so our responder emulates TCP servers on ports 80, 443, and 34343. Ports 80 and 443 are assigned by IANA for http and https traffic; port 34343 is unassigned. The client port is randomly chosen at connection setup.

Segments sent from the initiator include commands to operate the responder. The default command is "just ack", and the responder sends back a pure ACK (no data). Another command is "echo headers". Fig. 2.1 illustrates how this command works.

The initiator transmits a crafted segment that includes bytes indicating this command in its payload. The responder replies with a segment that contains in its payload both the received headers and the headers of the reply. The client then compares the sent and received headers for both segments to detect middlebox interference. The last command is "don't advance ack". The responder does not advance the ACK number when it receives this command; instead it sends back an ACK with the first sequence number of the receiving segment. This command is used in only the retransmission test in Sec. 2.2.5.

2.1.3 Measurement Data

Our measurements target access networks, where ISPs deploy middleboxes to optimize various applications with the goal of improving the experience of the majority their customers. The core is mostly just doing "dumb" packet forwarding. Many contributors and we ran the TCPExposure client in a variety of access networks detailed below. Contributors are mainly from IETF community, related research projects, and our labs. We ran the server tool (the responder) in *sfc.wide.ad.jp*, a middlebox-free network operated by our

Table 2.2: Experiment Venues

Country	*Home*	*Hotspot*	*Cellular*	*Univ*	*Ent*	*Hosting*	Total
Australia	0	2	0	0	0	1	3
Austria	0	0	0	0	1	0	1
Belgium	4	0	0	1	0	0	5
Canada	1	0	1	0	1	0	3
Chile	0	0	0	0	1	0	1
China	0	7	0	0	0	0	7
Czech	0	2	0	0	0	0	2
Denmark	0	2	0	0	0	0	2
Finland	1	0	0	3	2	0	6
Germany	3	1	3	4	1	0	12
Greece	2	0	1	0	0	0	3
Indonesia	0	0	0	3	0	0	3
Ireland	0	0	0	0	0	1	1
Italy	1	0	0	0	1	0	2
Japan	19	10	7	3	2	0	41
Romania	1	0	0	0	0	0	1
Russia	0	1	0	0	0	0	1
Spain	0	1	0	1	0	0	2
Sweden	1	0	0	0	0	0	1
Switzerland	2	0	0	0	0	0	2
Thailand	0	0	0	0	2	0	2
U.K.	10	4	4	2	1	1	22
U.S.	3	4	4	0	4	2	17
Vietnam	1	0	0	0	1	0	2
Total	49	34	20	17	17	5	142

japanese colleagues.

From 25th September 2010 to 30th April 2011, we measured 142 access networks in 24 countries. Table 2.2 shows the venues and the network types of the experiments.

Access networks are categorized in six types by human annotation. Home networks consisting of a consumer ISP and a home-gateway are labeled as *Home*. Public hotspots for example in cafes, airports, hotels, and conference halls are labeled as *Hotspot*. Mobile broadband networks such as 3G and WiMAX are labeled as *Cellular*. Networks in universities are labeled as *Univ*. We count two different networks (e.g., the lecture and the residence segments) in the same university as two university networks. Enterprise networks (also including small offices) are labeled as *Ent*. Networks in hosting services are labeled as *Hosting*.

2.2 Tests and Results

2.2.1 TCP Option Tests

TCP Options are the intended mechanism by which TCP can be extended. Standardized and widely implemented options include Maximum Segment Size (MSS), defined in 1981; Window Scale, defined in 1988; Timestamp, defined in 1992; and Selective Acknowledgment (SACK), defined in 1996. IANA also lists TCP options defined since 1996, but SACK is the most recently defined option in common use, and predates almost all of today's middleboxes. The question we wish to answer is whether it is still possible to rapidly deploy new TCP functionality using TCP options by upgrades purely at the end systems.

Unknown TCP options are ignored by the receiving host. A TCP extension typically adds a new option to the SYN to request the new behavior. If the SYN/ACK carries the corresponding new option in the response, the new functionality is enabled. Middleboxes have the potential to disrupt this process in many ways, preventing or at least delaying the deployment of new functionality.

If a middlebox simply removes an unknown option from the SYN, this should be benign—the new functionality fails to negotiate, but otherwise all is well. However, removing an unknown option from the SYN/ACK may be less benign—the server may think the functionality is negotiated, whereas the client may not. Removing unknown options from data packets, but not removing them from the SYN or SYN/ACK would be extremely problematic: both endpoints would believe the negotiation to use new functionality succeeded, but it would then fail. Finally, any middlebox that crashes, fails to progress the connection, or explicitly resets it would cause significant problems.

To distinguish possibly problematic behaviors, we performed the following tests:

(i) **Unknown option in SYN.** The SYN and SYN/ACK segments include an unregistered option.

(ii) **Unknown option in Data segment**. The test includes unknown options in data segments sent by client and server.

(iii) **Known option in Data Segment.** The test includes a well-known option in data segments sent by client and server.

All three tests are performed using separate connections. We do not use the unknown option in SYN for test 2 and 3. Test 3 is included to allow us to determine whether it is the unknown nature of the option that causes a behavior, or just any option. We use an MP_CAPABLE option for test 1 and an MP_DATA option for test 2; both options are defined in a draft version of MPTCP [27] and neither is currently registered with IANA, and no known middlebox yet supports them. On receipt of a SYN with MP_CAPABLE, our responder returns a SYN/ACK also containing MP_CAPABLE, and on receipt of a data segment with MP_DATA, it returns an ack packet containing an MP_ACK option, mimicking an MPTCP implementation.[1]

For test 3, we used the TIMESTAMP option [35], which is not essential to TCP's functionality, but which is commonly seen in TCP data segments. This option elicits a response from the remote endpoint; a stateful middlebox may also respond, allowing us to identify such middleboxes.

In the *unknown option in SYN test*, our code tests for the following possible middlebox behaviors:

- SYN is passed unmodified.
- SYN containing the option is dropped.
- SYN is received, but option was removed.
- Connection is reset by the middlebox.

[1]We use March 2010 draft version of these options' formats; MP_CAPABLE is 12 Byte length, MP_DATA is 16 Byte length, and MP_ACK is 10 Byte length. Option numbers are 30, 31 and 32, respectively.

Table 2.3: Unknown Option in Syn

Observed		TCP Port	
Behavior	34343	80	443
Passed	129 (96%)	122 (86%)	133(94%)
Removed	6 (4%)	20 (14%)	9 (6%)
Changed	0 (0%)	0 (0%)	0 (0%)
Error	0 (0%)	0 (0%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

Table 2.4: Known Option in Data

Observed		TCP Port	
Behavior	34343	80	443
Passed	129 (96%)	122 (86%)	133 (94%)
Removed	6 (4%)	9 (6%)	6 (4%)
Changed	0 (0%)	4 (3%)	3 (2%)
Error	0 (0%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

Table 2.5: Unknown Option in Data

Observed		TCP Port	
Behavior	34343	80	443
Passed	129 (96%)	122 (86%)	133(94%)
Removed	6 (4%)	13 (9%)	9 (6%)
Changed	0 (0%)	0 (0%)	0 (0%)
Error	0 (0%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

In the *unknown* and the *known option in data* tests, we test for the same behaviors as in the SYN test. After a normal handshake, the initiator transmits a full-sized TCP segment including MP_DATA or TIMESTAMP, using the "echo headers" command described in Sec. 2.1.2 to identify what the responder received. With this method we can identify which outbound or inbound option is interfered and whether the option is modified or zeroed. We also look for middleboxes that split the connection, processing the TIMESTAMP at the middlebox on either the inbound or outbound leg.

Middlebox Behavior on TCP Options

Tables 2.3 – 2.5 summarize the results of the options tests. 142 paths were tested in total; for ports 80 (http) and 443 (https), we obtained results from all paths for all tests. However seven paths did not pass the unregistered port 34343, even with regular TCP SYN segments. These paths appear to run strict firewall rules allowing only very basic services.

Most of the paths we tested passed both known and unknown TCP options without interference, both on SYN and data packets. The results are port-specific though; 96% of paths passed options on port 34343, whereas only 80% of paths passed options on port 80. This agrees with anecdotal evidence that http-specific middleboxes are relatively common.

All the paths which passed unknown options in the SYN also passed both known and unknown options in data segments. In the tables, the "*Removed*" rows indicate that packets on that path arrive with the option removed from the packet. For the unknown options in the SYN packet, this was the only anomaly we found;

no path modified the option or failed to deliver the packet due to its presence. In addition, all the paths which passed the unknown option in the SYN also passed unknown options in data segments. This bodes well for deployability of new TCP options—testing in the SYN and SYN/ACK is sufficient to determine that new options are safe to use throughout the connection.

Our test did not distinguish between middleboxes that stripped options from SYNs and those that stripped options from SYN/ACKs. With hindsight, this was an unfortunate limitation of our methodology that uses a stateless responder. However it is clear that any extension using TCP options to negotiate functionality should be robust to stripped unknown options in SYN/ACK packets, even if they are passed in SYNs. If it is crucial that the server knows whether or not the client received the option in the SYN/ACK, the protocol must take this into account. For example, TcpCrypt requires that the first non-SYN packet from the client contains the INIT1 option - if this is missing, TcpCrypt moves to the disabled state and falls back to regular TCP behavior.

For port 34343, the only behaviors seen were passing or removing options. The story is more complicated for port 80 (http) and 443 (https). There were seven paths that did not permit our testing methodology on port 80. In data packets our stateless server relies on instructions embedded in the data to determine its response. These seven paths appear to be application-level HTTP proxies, and we were foiled by the lack of a proper HTTP request in our data packets. They are labeled *Error* in the tables. We were able to go back and manually verify two of these paths were in fact HTTP proxies; we did not get a second chance to verify the other five. All seven were in the set that removed options from SYN packets, which is to be expected if they are full proxies. Two HTTP proxies that we manually verified removed options from data packets and resegmented TCP packets as well as proxies that are not HTTP-level ones.

There were no other unexpected results with unknown options, but we did observe some interesting results with the TIMESTAMP "known option in data" test. Four paths passed on a TIMESTAMP option to the responder, but it was not the one sent by the initiator. In these cases, although the responder sent TIMESTAMP in response, this was not returned to the initiator. This implies that the middlebox is independently negotiating and using timestamp with the server. These paths are labeled "*Changed*" in the tables. Paths in the *Removed* row in Table 2.5 correspond to those in the *Removed* or the *Changed* rows in Table 2.4 for all three ports. This implies that outbound option removal on data segments is not the unknown nature of the option.

Returning to the middleboxes that remove unknown options from the SYN, we can use the results of additional tests to classify these into two distinct categories. In the first category, the SYN/ACK received is essentially that sent by the responder, whereas in the second the SYN/ACK appears to have been generated by the middlebox. In Sec. 2.2.4 we explain how fingerprints in the SYN/ACK let us distinguish the two. Paths in the first category appear to actively eliminate options (we label them "*Elim*" in Table 2.6), whereas a middlebox in the second category is acting as a proxy, and unknown options are removed as a side effect of this proxy behavior (these are labeled "*Proxy*").

These two categories (*Elim* and *Proxy*) also hold when we look at data segments (see Table 2.7). Paths that

Table 2.6: Types of removal behavior (SYN)

Path Type	Other Observed Effects	TCP Port 34343	80	443
Elim.	None	5	4	5
Proxy	Proxy SYN-ACK	1	16	4
Total		6	20	9

Table 2.7: Types of removal behavior (Data)

Path Type	Other observed effects	TCP Port 34343	80	443
Elim.	None	5	4	5
Proxy	Proxy Data ACK, Segment Caching, Re-segmentation	1	9	4
Total		6	13	9

eliminate SYN options also eliminate data options, whereas paths that show proxy behavior on SYNs also exhibit proxy behavior for data. In particular, the proxy symptoms we see are Proxy Data Acks (Ack by the middlebox, see Sec. 2.2.4), segment caching (the middlebox caches and retransmit segments, see Sec. 2.2.5), and re-segmentation (splitting and coalescing of segments, see Sec. 2.2.6). These proxy middleboxes show symptoms of implementing most of the functionality of a full TCP stack, rather than just being a packet-level relay.

Before we ran this study, anecdotal evidence had suggested that cellular networks would be much more restrictive than other types of network. The results partially support this, as shown in Table 2.8. For port 80, eight out of 20 cellular networks that we tested remove options; six of the eight proxy the connection. WiFi hotspots are also relatively likely to remove options or proxy connections, especially for http. Overall though, the majority of paths do still pass new TCP options.

We conclude that it is still possible to extend TCP using TCP options, so long as the use of new options is negotiated in the SYN exchange, and so long as fallback to regular TCP behavior is acceptable. However, if we want ubiquitous deployment of a new feature, the story is more complicated. Especially for http, there are a significant number of middleboxes that proxy TCP sessions. For middleboxes that eliminate options, it seems likely that very simple updates or reconfiguration would allow a new standardized option to pass,

Table 2.8: Option removal by Network Type

Network Type	Remove option (Proxy conn) port 34343	port 80	port 443
Cellular (out of 20)	4 (1)	8 (6)	4 (1)
Hotspot (out of 34)	1 (0)	6 (5)	4 (3)
Univ (out of 17)	0 (0)	3 (3)	0 (0)
Ent (out of 17)	1 (0)	3 (2)	1 (0)
Total	6	20	9

Table 2.9: Sequence Number Modification Test

	TCP Port		
Behavior	34343	80	443
Unchanged	126 (93%)	116 (82%)	128 (90%)
Mod. outbound	5 (4%)	5 (4%)	6 (4%)
Mod. inbound	0 (0%)	1 (1%)	1 (1%)
Mod. both	4 (3%)	13 (9%)	7 (5%)
Proxy (probably mod. both)	0 (0%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

assuming it were not considered a security risk. But for transparent proxies, the middlebox would not only need to pass the option, but also understand its semantics. Such paths are likely to be more difficult to upgrade.

2.2.2 Sequence Number Modification

TCP Selective Acknowledgement (SACK) [40] is an example of a TCP extension that uses TCP options that quote sequence numbers, in this case to indicate precisely which segments arrived at the receiver. How might middleboxes affect such extensions?

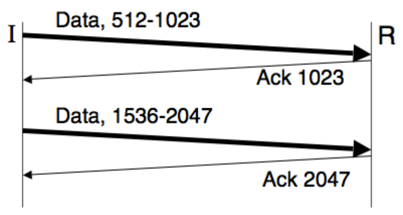
In our sequence number modification test, we examine both the outgoing and incoming initial sequence number (ISN) to see whether middleboxes modify the sequence numbers sent by the end systems. Table 2.9 shows the result. Paths where neither the outbound nor inbound sequence number is modified are labeled as *Unchanged*. Paths where the outbound or inbound sequence number is modified are labeled as *Mod. outbound* and *Mod. inbound*, respectively. Paths where both the outbound and inbound sequence numbers are modified are labeled as *Mod. both*.

Sequence numbers on at least 80% of paths arrive unchanged. However 7% of paths modify sequence numbers in at least one direction for port 34343 and 18% modify at least one direction for port 80. For port 80, the same seven paths identified earlier as having application-level HTTP proxies cannot be tested outbound, but do modify inbound sequence numbers and almost certainly modify both directions.

One might reasonably expect that middleboxes that proxy a connection would split a TCP connection into two sections, each with its own sequence space, but that other packet-level middleboxes would have no reason to modify TCP sequence numbers. If this were the case, then TCP extensions could refer to TCP sequence numbers in TCP options, safe in the knowledge that either the option would be removed in the SYN at a proxy, or sequence numbers would arrive unmodified. Unfortunately the story is not so simple.

At a TCP receiver, one use of sequence numbers is to verify the validity of a received segment. If an adversary can predict the TCP ports a connection will use, only the randomness of the initial sequence number prevents a spoofed packet from being injected into the connection. Unfortunately TCP stacks have a long history of generating predictable ISNs, so a number of firewall products try to help out by choosing a new more random ISN, and then rewriting all subsequent packets and acknowledgments to maintain consistency [33, 55].

We compared those paths that pass unknown options in the SYN with those that modify sequence numbers



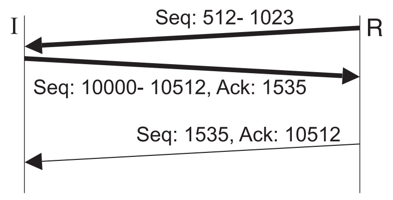


Figure 2.2: Sequence Hole Tests: data first (left) and ack first (right)

Table 2.10: Data-First Sequence Hole Test

	TCP Port		
Behavior	34343	80	443
Passed	131 (97%)	120 (85%)	135 (95%)
No response	2 (1%)	6 (4%)	2 (1%)
Duplicate Ack	1 (1%)	9 (6%)	5 (4%)
Test Error	1 (1%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

in at least one direction. On port 34343, 5 out of 9 allow unknown options and still modify the sequence numbers. For port 80, 7 out of 26 pass unknown options, and for port 443 it is 7 out of 14. The numbers are the same for unknown options in data packets.

We conclude that it is *unsafe* for TCP extensions to embed sequence numbers in TCP options (or anywhere else), even if the extension negotiates use via a new option in the SYN exchange.[2]

2.2.3 Sequence Space Holes

TCP is a reliable protocol; its cumulative Ack does not move forwards unless all preceding segments have been received. What would happen if from the vantage point of a middlebox, a TCP implementation violated these rules? Perhaps it wished to implement partial reliability analogous to PR-SCTP [50], or perhaps it simply stripes segments across more than one path in a similar manner to Multipath TCP?

We can distinguish two ways a middlebox might observe such a hole:

- **Data-First:** it sees segments before and after a hole, but does not see the segment from the hole. If the middlebox passes the segment after the hole, it sees it cumulatively acked by the recipient, despite the middlebox never seeing the data from the hole.

- **Ack-First:** It sees a segment of data, then an ack indicates the receiver has seen data not yet seen by the middlebox. If the middlebox passes the Ack, the next segment seen continues from the point acked, leaving a hole in the data seen by the middlebox.

These form the basis of our tests shown in Fig. 2.2. The left side is the initiator's time-line in both tests.

[2]SACK does embed sequence numbers in options, but it predates the existence of almost all middleboxes. We hope that these middleboxes are aware of SACK and either rewrite the options or explicitly remove SACK negotiation from the SYN exchange.

Table 2.11: Ack-first Sequence Hole Test

	TCP Port		
Behavior	34343	80	443
Passed	102 (76%)	95 (67%)	105 (74%)
No response	28 (21%)	28 (20%)	29 (20%)
Ack fixed	4 (3%)	5 (4%)	3 (2%)
Retransmitted	1 (1%)	7 (5%)	5 (4%)
Test Error	0 (0%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

Table 2.10 shows the result of the data-first sequence hole test. Paths where the second Ack was correctly received are labeled *Passed*, and clearly have no middlebox that requires TCP flow reassembly. As before, on port 80 there are seven paths with http proxies we cannot fully test; these are labeled *Test Error*. The one path using port 34343 labeled *Test Error* was due to high packet loss during the experiment rather than middlebox interference.

The remaining cases are the most interesting. We observed two distinct middlebox behaviors:

- *No response* was received to the second data packet.
- A *Duplicate Ack* was received, indicating receipt of the first data packet and by implication, signaling loss of the packet in the hole.

A middlebox implementing a full TCP stack would be expected to break the path into two sections, separately acking packets from the initiator before sending the data to the responder. This would give the *Duplicate Ack* behavior. As expected, we see more such middleboxes on port 80.

A middlebox that does not respond to the second packet is clearly maintaining TCP state (or it would pass the second Ack), but it is not independently acking data. Its reasons for doing so are unclear—perhaps it is attempting to analyze the stream contents and is unwilling to pass an ack for data it has not seen? Whatever the reason, we still see more such middleboxes on port 80.

In the ack-first sequence hole test (Fig. 2.2, right), the initiator acks a segment beyond that which is received (i.e., proactive ack). The responder skips the data acked and sends an ack packet the sequence number of which follows on from the point that was acked. To receive a response packet from the responder, the segment from the initiator to the responder also contains data, but what we are interested in is whether the proactive ack is received, and subsequently whether the packet following the hole is received. Table 2.11 shows the results.

The results of this test were a surprise—even on port 34343, middleboxes interfered with end-to-end behavior 24% of the time. As before, seven paths on port 80 could not be tested. Of those that could be tested, we saw three distinct behaviors:

- On around 20% of paths we saw *no response* to the proactive ack. Either the proactive ack was dropped or the packet above the hole was dropped, but the lack of a response does not allow us to distinguish.

- On quite a few paths (labeled *Ack fixed*), an ack packet is received, but the sequence number of which follows the last packet sent by the responder as if we sent an ack without holes. Perhaps the proactive ack was re-written by the outgoing middlebox to indicate the highest data cumulatively seen by the middlebox.

- On some paths, the middlebox itself actually *retransmitted* the last data packet sent by the responder from before the hole. These paths also sent back ack packets observed on *Ack fixed* paths, except for one path on port 80 and 443.

It is clear from these results that TCP extensions relying on sequence number holes are *unsafe*. Although some of the results can be explained by proxy behavior at middleboxes, some paths that did not exhibit clear proxy behavior (by performing separate acknowledgment) do affect both sequence holes and pro-active acking. Perhaps some firewalls attempt to protect the initiator from potentially malicious proactive acks? [48]. One interesting observation is that around 10% of home networks give *no response* in the ack-first sequence hole test. This is striking because none of the home networks strip unknown options.

2.2.4 Proxy Acknowledgments

In Tables 2.6 and 2.7 we observed that a subset of the paths that remove TCP options appear to show TCP proxy behavior. We now elaborate on the tests we used to elicit this information.

A hypothetical TCP proxy[5] would likely split the TCP connection into two sections; one from the client to the proxy and one from the proxy to the server. Each section would effectively run its own TCP session, with only payload data passed between the two sections. Are the proxies we observed of this form, which is fairly easy to reason about, or is their behavior more complex?

One symptom of a TCP proxy would be that acknowledgments for data are locally generated by the middlebox. We performed two tests examining this behavior:

- **Proxy SYN-ACK:** Is the SYN/ACK locally generated by the proxy? In its SYN/ACKs, our responder generates quite characteristic values for the initial sequence number, advertised receive window, maximum segment size, and Window Scale options. It is improbable that a proxy would generate these values. We simply check the value of these fields in the SYN/ACK received by the initiator—if they differ then this is symptomatic of a proxy that crafts its own SYN/ACKs.

- **Proxy Data Ack:** Is data acknowledged by the proxy before delivering it to the destination? Our initiator sends a data packet to the responder, requesting the ack is sent on a packet that includes data. If the ack received does not include data, it is extremely likely it was generated by the proxy rather than the responder.

Neither test is conclusive by itself, but taken together they give a good picture of proxy behavior. As before, there are seven paths which have HTTP-level proxies; on port 80, all seven sent proxy SYN/ACKs, but could

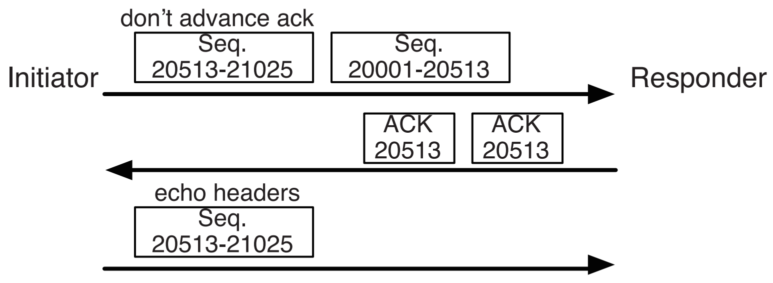


Figure 2.3: Retransmission Test

Table 2.12: Results of Retransmission Test

Observed Behavior	TCP Port / Retransmitting size								
	34343			80			443		
	same	smaller	larger	same	smaller	larger	same	smaller	larger
Passed	134 (99%)	134 (99%)	132 (98%)	124 (87%)	124 (87%)	123 (87%)	138 (97%)	138 (97%)	136 (96%
No response	0 (0%)	0 (0%)	1 (1%)	0 (0%)	0 (0%)	1 (1%)	0 (0%)	0 (0%)	1 (1%)
Ack adv'ced	1 (1%)	1 (1%)	1 (1%)	10 (7%)	10 (7%)	10 (7%)	4 (3%)	4 (3%)	4 (3%)
Reset conn	0 (%)	0 (0%)	0 (0%)	1 (1%)	1 (1%)	1 (1%)	0 (0%)	0 (0%)	0 (0%)
Error	0 (0%)	0 (0%)	1 (1%)	7 (5%)	7 (5%)	7 (5%)	0 (0%)	0 (0%)	1 (1%)
Total	135 (100%)			142 (100%)			142 (100%)		

not be tested for proxy data acks. Tables 2.6 and 2.7 show the number of proxies identified. The set of paths showing Proxy SYN/ACK behavior is precisely the same as those showing either Proxy Data Ack or HTTP proxy behavior. Taken together, these tests provide good evidence for proxies of the form described above.

2.2.5 Inconsistent Retransmission

If a TCP sender retransmits a packet, but includes different data than the original in the retransmission, what happens? This might seem like a strange thing to do, but it might be advantageous for extensions that do not need stale data (such as VoIP over TCP). Given that we know sequence holes are a bad idea (See Sec. 2.2.3), it might make sense to fill the sequence hole with previously unsent data.

Such inconsistent retransmissions would be explicitly "corrected" by a traffic normalizer[33], as its role is to ensure that any downstream intrusion detection system sees a consistent picture. Equally, depending on their implementation, TCP proxies might reassert the original data. We set out to test what happens in reality.

Fig. 2.3 shows our retransmission test. The initiator sends two consecutive segments, but we request that the responder sends a cumulative ack only for the first segment, then a duplicate Ack. Any stateful middlebox will infer that the second segment has not been received by the responder, and depending on its implementation, it may retain the unacked segment. We then send a "retransmission" of the second packet, but with a different payload (one that requests the responder echo the packet headers so we can see what is received).

We also repeat the test, but with the "retransmitted" packet being either 16 bytes smaller or 16 bytes longer than the original packet.

From the responses, we can distinguish four distinct middlebox behaviors, as listed in Table 2.12:

- Most paths *passed* the inconsistent retransmission to the responder unmodified. In the case of port

34343, only one path did not do this.

- On some paths the initiator observes that the cumulative *Ack advanced*, but the headers were not echoed. This implies that the middlebox cached the original segment and resent it. Most of these paths were ones that we had previously identified as TCP proxies, but one on port 80 was not—it caches segments but does not separately ack data. We cannot know for sure, but this would be symptomatic of a traffic normalizer or a snoop [6]. For port 443, one path in fact echoed headers after the separate cumulative ack packet for the retransmission of the 16 byte longer packet. However, what the responder received is a 16 byte piece that does not overlap with the original—the other part is probably cached by the middlebox.
- One path returned *no response* at all when the inconsistent retransmit was larger than the original, and did so for all ports. There is no obvious reason for such behavior, so we speculate it might be a minor bug in a middlebox implementation.
- One path on port 80 *reset the connection*. This seems to be a fairly draconian response.

The usual seven paths with HTTP proxies could not be tested. One path on port 34343 and one on port 443 also failed to complete the test due to high packet loss.

Overall, any extension that wished to use inconsistent retransmissions would encounter few problems, so long as it did not matter greatly whether the original or the retransmission actually arrives. The one path that resets connections might however give the designers of extensions cause for concern.

We note that the proposal for TCP extended options might result in retransmissions that appear inconsistent to legacy middleboxes, even if the payload is consistent. This might occur if the value of an extended option such as a selective acknowledgment changes between the original and the retransmission.

2.2.6 Re-segmentation

TCP provides a reliable bytestream abstraction to applications, and makes no promises that message boundaries are preserved. Some TCP extensions such as TcpCrypt wish to associate a new option with a particular data segment—in the case of TcpCrypt to carry a MAC for the data. How will such extensions be affected by middleboxes?

We expect that TCP proxies will coalesce small segments if a queue builds in the proxy, and might split segments if the proxy negotiates a larger MSS with the client than that negotiated by the server. However, our results show such proxies remove unknown options from the SYN exchange, so any adverse interaction (beyond falling back to regular TCP) is unlikely. Our concern therefore is whether there are middleboxes that are not proxies that re-segment packets. In particular, any middlebox that passes new options *and* also re-segments data might be problematic.

To test segment splitting, we simply send a full-sized segment. Our responder advertises a relatively small 512 byte MSS. Any middlebox advertising a more normal (larger) MSS will be forced to resegment larger

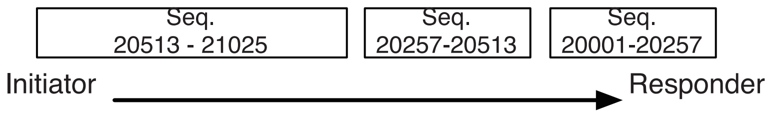


Figure 2.4: In-order Segment Coalescing Test

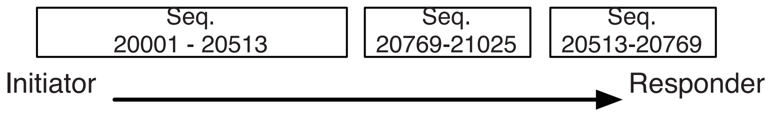


Figure 2.5: Queued Segment Coalescing Test

data packets into smaller ones. In fact, MSS advertised by 16 SYN proxies we observed at port 80 varied between 1372 – 1460 bytes. We perform the test without option, that with the known option (TIMESTAMP) and that with the unknown option (MP_DATA) to see if options are copied to the split segments.

We found that 1 path on port 34343, 9 paths on port 80 and 4 paths on port 443 split segments in this way. These are the same paths identified as proxies in Table 2.7. None passed options to the split segments.

The opposite of segment splitting is segment coalescing, where a middlebox combines two or more segments into a larger segment. To test for this, we must send two consecutive small segments and observe whether a single larger segment arrives. However, a middlebox that has the ability to coalesce might still not do so unless it is forced to queue the segments. We therefore perform two versions of the test, as shown in figures 2.4 and 2.5.

- We test if segments are coalesced if the two small segments arrive in order (Fig. 2.4).
- We reorder the segments so that the small segments arrive after a gap in the sequence space, creating an opportunity for middleboxes to queue them (Fig. 2.5). We then send the segment which fills the sequence hole. If a middlebox queued the small segments, this will release them, potentially allowing coalescing to occur.

As before, we repeat the tests without options and with both known and unknown options.

Table 2.13 shows the results. Most middleboxes running TCP proxies coalesced segments in both in-order and queued cases (labeled *Coal. both*), and the other proxies did so in only the queued case (labeled *Coal. queued*). No middlebox copies either known or unknown options to the coalesced segments. One non-proxy path did coalesce segments in the in-order test on ports 80 and 34343 (labeled *Coal. ordered*), but passed all the other tests. Interestingly, it only coalesced when options were not present.

As before, on port 80 seven HTTP proxy paths could not be tested. Three other cases gave unexpected results. One path on port 34343 failed in the queued test that does not contain options, but did not coalesce in the other tests. One path on port 80 acked only the third segment in the queued test—returned no payload; other tests show this path does not show proxy behavior and does pass TCP options, but gives no reply to the data-first sequence hole. Likely it is also ignoring out of order segments in this test too. The other path on port

Table 2.13: Results of Segment Coalescing Test

Observed	TCP Port		
Behavior	34343	80	443
Passed	132 (98%)	123 (87%)	138 (97%)
Coal. ordered	1 (1%)	1 (1%)	0 (0%)
Coal. queued	1 (1%)	3 (2%)	1 (1%)
Coal. both	0 (0%)	6 (4%)	3 (2%)
Error	1 (0%)	9 (6%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

80 showed similar behavior except that it does not return payload even in the in-order test and does cache segments. We do not know what form of middleboxes these are, but their behavior seems fragile.

Among those paths that coalesced, we saw quite a variety of behavior. The two small segments we sent were of 244 bytes. When coalescing occurred, depending on the path, the first coalesced segment received could be of 256, 488, 500 or 512 bytes in the first test and 256, 476 or 488 bytes in the second test. We have no idea what motivates these particular segment sizes.

Overall, the story is quite good for TCP extensions. Although middleboxes do split and coalesce segments, none did so while passing unknown options (indeed one changed its behavior when options were present). Thus it seems relatively safe to assume that if an option is passed, it arrives with the segment on which it was sent.

2.2.7 Intelligent NICs

Most of the experiments in this paper probe the network behavior, but with the rise of "intelligent" Network Interface Cards, even the NIC can have embedded TCP knowledge. Thus the NIC itself might fight with new TCP extensions.

We are concerned in particular with TCP Segmentation Offload (TSO), where the host OS sends large segments and relies on the NIC to resegment to match the MTU or the receiver's MSS. In Linux, the TCP implementation chooses the split segment size to allow all the TCP options to be copied to all the split segments while still fitting within the MTU. But what do NICs actually do—do they really copy the options to all the split segments?

We tested twelve TSO NICs from four different vendors; Intel (82546, 82541GI, 82566MM, 82577LM, 82567V, 82598EB), Nvidia (MCP55), Broadcom (BCM95723, BCM5755) and Marvell (88E8053, 88E8056, 88E8059). For this, our initiator tool consists of a user application and a custom Linux kernel, and we reused the responder tool from the earlier middlebox tests. The key points about the experiment are:

- Our application calls write() to send five MSS of data to the socket layer at one time.
- The OS TCP stack composes one TCP segment that includes all the data and passes it to the TSO layer. This large segment also includes the TIMESTAMP or MP_DATA TCP option.

- The NIC performs TSO, splitting the large segment into multiple segments and transmits them.
- Our responder receives these segments and responds with a segment echoing the headers in its payload so we can see what was received.

All the NICs we tested correctly copied the options to all the split segments. TSO is now sufficiently commonplace so that designers of extensions to TCP should assume it. The implication is that TCP options must be designed so that when they are duplicated on consecutive segments, this does not adversely affect correctness or performance.

We also tested Large Receive Offload (LRO) behavior with the Intel 82598EB ten gigabit ethernet NIC to see how TCP options are treated. First, we receive bulk TCP traffic with the NIC; all packets in the traffic include an MP_DATA option with the same values. Second, we receive similar traffic, but change the values of the MP_DATA between packets. We also conducted the same tests with a TIMESTAMP option instead of the MP_DATA. For both option kinds, packets were coalesced only when their option values are same. The coalesced segment has one of the options on the original segments. This behavior seems sane: on this particular NIC, LRO simply tries to undo what TSO did by duplicating options. If options are different, no coalescing happens.

Both TSO and LRO seem to forbid TCP extensions to reliably use the counts of sent and received options for signaling. Instead, TCP extensions experiencing offload should be prepared to handle both duplicate and "merged" options. Disabling offload altogether at endpoints is possible, but will result in a performance penalty.

2.3 Lessons learned

In the original TCP specification, the options were included to allow TCP extensions to be incrementally deployed. A typical extension defines a new TCP option and places it in the SYN segment. If the SYN+ACK contains a similar option, the extension is used. Otherwise, the extension is not enabled. This option-based extensibility worked well in a network that followed the end-to-end principle. Unfortunately, in a network that contains middleboxes, this is not sufficient.

In today's Internet, the three-way-handshake involves not only the two communicating hosts, but also all the middleboxes on the path. Verifying the presence of a particular TCP option in a SYN+ACK is not sufficient to ensure that a TCP extension can be safely used. As shown in the previous chapter, some middleboxes pass TCP options that they don't understand. This is safe for TCP options that are purely informative (e.g. RFC1323 timestamps) but causes problems with other options such as those that redefine the semantics of TCP header fields. For example, the large window extension in RFC1323 changes the semantics of the window field of the TCP header and extends it beyond 16 bits. Nearly 20 years after the publication of RFC1323, there are still stateful firewalls that do not understand this option in SYNs but block data packets that are sent in the RFC1323 extended window. A TCP extension that changes the semantics of parts of the packet header must include mechanisms to cope with middleboxes that do not understand the new semantics. Our own experience in designing Multipath TCP, an extension to TCP described in detail in Deliverable 4.3, has brought up many design issues that do not exist in an end-to-end Internet. A major issue we encountered affects all TCP designers and not just MPTCP. This issue is *the mutability of the TCP packets*. In an end-to-end Internet, all the information carried inside TCP packets is immutable. Today this is no longer true. The entire TCP header and the payload must be considered as mutable fields. If a TCP extension needs to rely on a particular field, it must check its value in a way that cannot be circumvented by middleboxes that do not understand this extension. Multipath TCP includes an additional checksum (the DSM checksum) in every packet just to deal with the problem of mutable packets.

The third, but probably most important point about new TCP extensions is that to be deployable they must necessarily include techniques that enable them to fallback to regular TCP when something wrong happens. If a middlebox interferes badly with a TCP extension, the problem must be detected and the extension automatically disabled to preserve the data transfer. A TCP extension will only be deployed if its designers can guarantee that it will transfer data correctly (and hopefully better) in all the situations where a regular TCP is able to transfer data.

The last point that we would like to raise is that the hidden middleboxes increase the complexity of the network. This gives us a strong motivation to change the network architecture to recognize explicitly their role. The CHANGE architecture aims to do precisely this, embracing flow-processing and implicitly middleboxes, while allowing the Internet to evolve at the same time.

3 CHANGE Architecture

Whether we would like it or not, there is no returning to the original end-to-end transparent Internet architecture. Quite simply, there are just too many reasons why processing of data flows needs to be performed within the network, not just in the end-systems. To enable innovation, we need to play to the strengths of both packet-switching and flow processing, rather than dogmatically camping on one side or the other. We acknowledge the fundamental advantages offered by packet-switching, but believe that the architecture is missing the primitives to introduce and reason about flow-processing at selected key points in the network. We assert that flow processing must not be provided in the form of in situ hacks (as is the current state of affairs) but in a way that makes it a first class object in the network. Done right, we believe it will allow operators and application writers to reason about the emergent behaviour of the end-to-end path through such a network - an arduous task in today's Internet.

The deployment of a general purpose flow-processing architecture is what is required to break the innovation log-jam that has been developing over the last fifteen years. This is the overall goal of the project. But before delving deeper, we must first answer the question; *what exactly is flow processing?*.

Flow processing is any manipulation of packets where the service given to or the operations applied to a set packets is differentiated based on their implicit membership of a labeled flow. Flows can be of different granularity; a single TCP connection might comprise a flow, but equally all the traffic between two sites can comprise an aggregate flow, or even all the Internet telephony traffic traversing a router. The concept of a flow is therefore generic, it simply defines a relationship between a set of packets. But what all flow processing has in common is that it requires the maintenance of some measure of flow state in the network.

The processing that might be applied to flows is very varied. At one extreme, flow processing might involve providing low-latency forwarding to traffic in a flow. At the other extreme, it might involve the reassembly of the contents of a packet stream and parsing of application-level content to perform network intrusion detection, or explicit authentication to a site firewall to allow subsequent packets of a flow to proceed.

To enable innovation, an architecture must be able to support a wide range of flow processing. It must allow for the rapid innovation and deployment of new flow processing primitives so that flow processors are not locked into the applications of today. In essence, this means that anything more than simple packet forwarding should be a software function, so as to allow for quick deployment. Contrast this with the current Internet, where flow processing is almost always performed in special purpose boxes sold by vendors to solve a specific problem.

In recent years several trends have come together to transform this general vision and bring it to practical reality. First, general-purpose x86 server hardware has become cheap and powerful enough for packet processing at rates of up to 20Gbit/s. The combination of PCI-Express, Gigabit or 10-Gigabit Ethernet supporting virtual queueing, CPUs with many cores, and high-bandwidth NUMA systems architectures, has resulted in low-cost systems that have been optimized for high-performance network processing on general purpose servers. Such machines are equally comfortable at performing network flow processing, and having been optimized for virtualization, individual machines are able perform flow processing on behalf of more than one organization. Secondly, high-performance Ethernet switch chipsets have become a low-cost commodity item. Such chipsets have flow-processing capabilities; typically they are able to perform matching and forwarding based on arbitrary combinations of packet header fields, as well as supporting multipath forwarding; useful for hash-based load-balancing across multiple output ports. Current commodity chipsets can support flow tables containing tens of thousands of flows. OpenFlow [41] takes advantage of such chipsets and provides a common API to control flow processing in these switches.

Combining OpenFlow-style switches with clusters of commodity servers allows powerful and scalable platforms to be built [30]. The switch provides the first level of classification, and balances traffic directly across virtual queues on the servers, allowing traffic to be directed to specific CPU cores for more sophisticated processing. Such a hardware platform provides a very powerful, scalable and flexible base on which to build a flow processing system. An enabling goal of CHANGE is to build upon preliminary work on these platforms, with the aim of realising flow processing systems that can take full advantage of the flexibility inherent in such a hardware platform.

Flow processing platforms offer the potential to realise several distinct advantages:

- The capacity to scale up processing by merely adding more inexpensive servers
- The capacity to scale down processing by concentrating load on a few boxes at quiet times to save power consumption
- The capacity to roll out new flow processing functionality at short notice to handle unexpected problems, or take advantage of unexpected opportunities, with only software reconfiguration required.
- The capacity to support a wide range of functionality thanks to relying on general-purpose hardware and operating systems
- The capacity to dynamically shift processing between flow processing servers
- The capacity to concurrently run different kinds of processing on different sets of flows while providing high performance and fairness guarantees

Running isolated CHANGE platforms brings many benefits to the deploying entity: it will be easy to update processing functionality and to increase processing capacity on demand. However, this is only a small improvement with regards to both the transparency and the rapid evolution of the Internet. The real benefits come when the CHANGE platforms cooperate to ensure end-to-end flow processing, at which point users will be able to reason about expected performance of their instantiations, detect bottlenecks, and scale out processing, all in a principled and transparent manner.

3.1 Architecture Requirements

The Internet is a playing field where different stakeholders fight to impose their requirements. End-hosts wanting to run new applications would like the Internet to return to the days of ubiquitous reachability (that allows peer-to-peer operation, for instance) and a "dumb" network that treats all packets equally. The network operators, on the other hand, must enforce security via firewalls, deal with the depleting IPv4 address space by deploying NATs, and use DPI to differentiate traffic to impose some notion of "fairness" among users. These tussles in cyberspace will not be resolved any time soon, they are built in to the fabric of the Internet [20]. Therefore technology must not take sides, and solutions should allow for the tussle to play out in different ways. Currently the balance of power is tilted towards the middle; ISPs deploy functionality that makes it harder and harder for the endpoints to deploy new applications, and applications get more complex in order to jump the hurdles.

We have talked at great length about what MPTCP needs to do to get through the current Internet in the previous chapter. While it does work, the problem with MPTCP is that its defensive design carries a certain amount of overhead, for instance adding checksums to each packet. If the network evolves to be friendlier in the future, MPTCP won't really be able to tell and will still be as defensive as today when doing its functionality. The network could help, but there is no way today for the ends to communicate with the middle. That needs to change.

Skype is another telling example: it does almost everything possible to get through NATs and firewalls, including tunneling voice traffic over TCP and using application-level proxies to allow conversations between two hosts behind NATs. Anecdotally, a big part of Skype's success stems from the very fact that it works most of the time where no other similar app achieves quite the same results.

While Skype does work, it is far from optimal: voip traffic between two users in the same city can sometimes end up re-routed through different countries, inflating end-to-end RTT and decreasing user satisfaction. This state of affairs is suboptimal for the operators too: they have to pay for inter-domain traffic that should have never left their network.

In fact, this is a symptom of a bigger problem, and Skype is not an isolated case: iPlayer and BitTorrent are examples of other applications whose traffic can take suboptimal paths that hurt the operators' economic interests. The real problem is that the operators' policy is unknown to the applications, and the applications' desires are unknown to the operators. The result is an escalating war where DPIs are deployed and P2P traffic is throttled, then the apps encrypt the traffic to avoid DPI, and so on.

To avoid all apps or endpoints having to include complex functionality just to ensure basic functionality such as reachability, the network must be able to provide, on request, standard functionality that makes it possible for endpoints to communicate even if they are behind NATs. One useful functionality is the the ability to encapsulate and decapsulate packets in the network. Further, the users must be able to tell the network what they want and the network should help them implement that processing, rather than placing all the processing at endpoints - this will allow us to avoid the suboptimal routing we saw earlier and also help design a simpler MPTCP.

All of these can be accomplished within a flow processing architecture, as proposed by the CHANGE project. The original Internet architecture lacked the concept of a flow, except as realized at the end systems. The end-to-end principle provides a good argument against putting functionality in the core of the network, but taken to its extreme it means than a network operator cannot see if the network is actually functioning. A network can be moving very large numbers of packets but still getting no useful work done, as in the case of a DDoS attack. Although packets are the basic forwarding entity, it is flows that perform work, and so it should be unsurprising that network operators insert equipment into the network to manage and enhance flows.

Retrofitting a flow-processing abstraction to the Internet architecture is difficult to do in a clean way. With hindsight, the original placement of addresses in IP and ports in TCP and UDP might have been a mistake, as it places at least one component of flow identification in L4, where it was not intended to be used to make forwarding decisions. Nevertheless, we assert that it can be done. The main pieces are [1]:

- A general-purpose scalable flow processing platform on which L4+ middlebox functionality can be implemented in software and updated as apps change.
- A categorization of flow processing behavior into a handful of classes, so we can reason about the behaviour of concatenations of middleboxes without needing to understand the precise functionality of each.
- A way to identify who can request processing.
- A way to name flows.
- A way for end-systems to discover platforms close to them or to the paths that their flows take.
- Attraction of flows to platforms that will process them.

The requirements above are functional, saying what CHANGE should incorporate to be able to fulfill its tasks. Given these, how should we choose from the wide range of implementation possibilities that exist? Our goals are to create an architecture that is feasible to deploy and use in practice and is appealing to both operators and end-users. In the next section we discuss a few high-level requirements that steer CHANGE towards these goals.

3.2 End-to-end example

To give the reader a crisper understanding of what CHANGE is trying to achieve, we describe here an end-to-end example of CHANGE functionality along with the way it is implemented. We will spend the rest of this document explaining the whole architecture in greater detail.

[1]These components follow directly from the functional architecture requirements spelled out in Deliverable 2.1

Consider a user wanting to instantiate an in-network Intrusion Detection System to analyze traffic before it reaches its premises. The user will create a CHANGE configuration file that describes the processing it wants instantiated. The configuration file allows the user to specify how its packets should be processed on CHANGE platforms. It allows platform selection, flow naming, wide-area flow manipulation (redirection, rerouting, tunneling) as well as arbitrary flow processing.

The software running at the client will parse the configuration file and implement it. First, the user might specify that it wants a CHANGE platform close to him (in terms of network delay), and this will be implemented by using mechanisms akin to CDNs replica selection, or by using anycast as we propose in Chapter 3.7. The client software will then contacts the platform and prove it owns its address by using the newly deployed RPKI infrastructure. This convinces the platform that it is safe to run processing on traffic destined to the user. At this point, an extra step may be needed if the traffic does not naturally pass through the selected CHANGE platform: DNS or BGP could be used to draw the traffic into the platform. The IDS is then instantiated using the rules provided by the user. To ensure that traffic is not changed between the platform and the destination, the user specifies a network invariant requesting that existing packets cannot be modified and that new packets cannot be originated with the same destination address. Finally, the user pays the platform owner for the processing it is providing, which provides incentives for ISPs to deploy such platforms.

3.3 Flow Naming

We begin our discussion of the CHANGE architecture by asking the most basic question: *what is a "flow"*? We need a definition that allows both users and platforms to uniquely identify the packets belonging to a single flow, both for labeling and processing.

The answer is simple for a single platform: a flow is any subset of the packets passing through that platform. In this context, then, the question becomes how can a user specify a subset of packets. CHANGE uses a bitmask for this purpose; we discuss the reasons for this choice in some detail in section 3.3.1. For instance, a user could run an intrusion detection system on platform P1 analyzing all the traffic addressed to it (as in our end-to-end example discussed above).

The next question is: how do we define a flow in a wide-area context? Intuitively, we want to be able to identify on downstream platforms *exactly the same packets* selected at the ingress CHANGE platform(s). An ideal solution would be to insert unique, immutable tags in packets specifying the flow they are part of. The great virtue of this solution is that flow names are explicit and visible throughout the Internet. However, in practice this solution is infeasible because we cannot implement the required in-band signaling in a deployable way in the current Internet.

The CHANGE architecture uses *implicit* flow definitions instead: these cannot be seen by observing the packets. When it specifies a CHANGE configuration, the user can define a flow as the output of filtering input packets with bitmaps or as the output of processing modules in the CHANGE platform. The user can then use the flow name to request processing of that flow on a different CHANGE platforms. To implement this functionality, CHANGE uses:

- Bitmasks on each platform to specify which packets belong to each flow. Given the original packet flow, CHANGE computes new bitmasks that will match the same packets on another platform, and it can forward the packets to that platform if needed. By itself, the bitmasks are not enough as packets could be changed in transit or new packets could be injected into the flow.
- To avoid this problem, invariants are used to guarantee that the selected set of packets is indeed the same as that seen on the ingress platforms.

We discuss in greater detail the reasoning that lead us to this solution in section 3.3.2.

3.3.1 From Packets to Flows

At the highest level of abstraction, flow identification is equivalent to defining a Boolean function, such that given a flow definition and an arbitrary packet, it returns $TRUE$ iff the given packet belongs to the named flow and $FALSE$ otherwise.

In principle, there are two basic requirements for a flow naming solution; *flexibility* and *performance*. The flexibility clause is best expressed when we look at how a flow definition may be used. For example, a flow definition may be required to capture a single TCP connection, or all UDP traffic destined to an IP address, or ICMP traffic originating from a given source. From these examples it becomes clear that the range of possibilities is clearly vast, and as such an adequate flow naming solution must be flexible enough to capture all these flows.

Given that platforms will capture the flows of interest to them by applying filters to the transient traffic they see, it is obvious that this fundamental task must be carried out efficiently. To realise this, flow naming must be amenable to a fast, line-speed implementation. This performance requirement is somewhat elusive: though hardware advances will push the envelope for what processing capacity, it is not adequate to simply rely on “better hardware in the future”; instead, the basic definition presented must be implicitly suited to high-performance processing.

Conventional wisdom holds that maintaining and accessing per-flow state for all Internet traffic is in practice unfeasible. The reasons for this are mostly economic; while it is possible to build routers with large and fast flow memory in practice these are prohibitively expensive. Though DRAM-based solutions are cheaper, they are also slower. To allow fast identification of a flow, we require that a flow-processing platform be stateless; it does not maintain additional per flow state. This means the Boolean function identifying a filter must also be memory-less so as not have any side effects.

Traditionally, flows are defined using the 5-tuple syntax of `(IP_SRC, PORT_SRC, IP_DST, PORT_DST, PROTO)`, while the corresponding Boolean function implements a check for the exact matching of all the five fields. The most evident disadvantage of the 5-tuple syntax is that it cannot scale up and down the protocol stack. For example, if a flow consists of all HTTP GET packets to one destination, there is not a single function that matches all the desired packets.

A more scalable approach to the problem is to make the flow identification process protocol agnostic. By this we mean replacing the implicit IP header dependence of the traditional 5-tuple syntax with an easily generalisable *bitmask*, such that if the bitmask matches a packet, the packet is labeled as belonging to the flow that defines the bitmask.

Such a bitmask could be used to address any bit from an IP packet, including payload. For example, given the first byte of 4 packets as: `0110 0110`, `0111 1010`, `1010 0110` and `1110 0001` and the bitmask: `⋆1⋆0 ⋆⋆⋆⋆` - whereby $*$ defines a wild card, it is clear that AND'ing the bitmask will only match packets 1 and 4.

Because of the generality of this approach, it is trivial to craft bitmasks that can support basic flow identification tasks such as:

- Identifying a TCP connection - specify the 5-tuple
- Identifying ICMP traffic originating from a given host- match the ICMP protocol number and the source address.
- Identifying the packets going to one destination host: we only match the IP_DST field.
- Identifying the packets destined for a group of hosts in a subnet: we simply modify the mask, in order to contain all the hosts from that group. (e.g. match 128.16.6.*)

Matching bitmasks has the advantage that is easy to implement in hardware using TCAM memory - which is becoming cheaper and increasingly commonplace with the recent advent of OpenFlow switches. While our flow definition is naturally applicable to packet headers and allows for expressive selection of flows at layers two, three and four, be they host-to-host, many-to-one or even many-to-many. Further, bitmasks can be used to select application level flows where port numbers are sufficient for identification.

Though immensely flexible, there exists a whole class of application-level flow definitions (characterised by the requirement that flow state information be kept) that our solution does not capture. For instance, selecting flows that contain a certain worm signature or HTTP requests for a given Youtube video or a the packets associated with a given FTP session or all the sub-flows belonging to a given multipath TCP connection.

In principle some of this functionality can be achieved by applying the bitmasks on packet payloads. For instance, if the destination IP is that of Youtube and the payload is `'GET /video.avi'` we could redirect the traffic to a local cache. However, there are implicit limitations with such an approach. For example:

(i) Packets belonging to a flow may get redirected while others don't - in the Youtube example, the request segment is redirected but the SYN exchange or subsequent packets are not. This is not what we would like to happen: we would like the whole flow to be redirected, or at least the part of flow after the request.

(ii) Middleboxes could split/re-segment packets, which would make payload bitmasks fail. In the Youtube example, if the resource name is split across two segments the bitmask will fail to match the segments.

(iii) Interesting patterns could start at different offsets within the payload (e.g. a worm signature).

The first problem can be fixed if there is some callback for a matched packet (e.g. redirect to rule owner), whereby a new rule would be inserted when a match occurs. For example, in order to capture the Youtube cache, a new 5-tuple rule would be inserted to match all subsequent packets associated with the flow.

The second and third problems are more difficult to solve, because they require the flow processing platforms to reconstruct the TCP payload before they apply the matching. Further, the matching itself needs to be more complex to support varying offsets; (e.g., full blown regular expressions for searching worm signatures). The commonality between the listed examples is that they require per flow state to function and as such will not scale to large numbers of flows.

As discussed, though limited, our bitmask based solution (applied to packet headers) provides the highest level of generality without instantiating per-flow state. Applications that do require per-flow state can still be supported, by using a two-stage approach to first identify and then handle their traffic. First the bitmasks are used to select a slice of the flows (i.e. flows going to a certain destination, or having a certain destination port, etc.) and then they are redirected to a local processing module. Second, the processing modules keep per flow state as required, and implement the desired processing functionality, for instance detecting flows that carry a given worm signature. Once detected, these flows will be either redirected for further processing (c.f. FlowStream), tunneled to the destination, dropped, etc.

Finally, while we consider a packet-level bitmask to uniquely define flows, there are currently limitations due to the OpenFlow architecture. OpenFlow currently accepts only bitmasks based on source IP and destination IP addresses, and exact matches on other L2 and L3 header fields, as well as TCP ports. It is expected that OpenFlow will in the future support bitmasks for arbitrary bits in the header.

3.3.2 Wide-Area Flow Processing

How do we define a flow in a wide-area context? A strawman solution is to consider that a flow comprises packets matching the *same* bitmask on many platforms in the Internet. However, this solution is not adequate for many reasons. First, packets change during forwarding and the same bitmask may not be able to capture the same packets as they pass through different platforms—an obvious example here is when the mask includes the TTL field. Even more conventional bitmasks, such as those based on addresses in packets, can fall short because of NAT boxes, proxies, and so forth. Even worse, using the same bitmask on different platforms could match different packets belonging to different users: a filter using non-routable addresses (192.168.x.y) would fall in this category, matching packets from different users and wrongly placing them in the same flow.

An ideal solution would allow us to *tag* the packets selected by the user on the ingress platform: it should be possible to add a globally-unique identifier to each packet such that a) any CHANGE platform observing the packet can tell which flow it belongs to and b) it is impossible for anyone except the ingress platform/user to create a valid packet identifier belonging to that flow.

Cryptographic techniques can be used to generate such identifiers. For instance, a flow could be uniquely defined by a public/private key-pair. To mark a packet as belonging to a flow, we would include in the packet a hash of the public key, and a signature of a predefined part of the packet (for instance, the payload). Then, any platform that is given a flow definition (the public-key together with the part of the packet that needs to be checked) can easily verify that the packet belongs to the flow by first matching the public key and then verifying the signature.

This definition is quite powerful: it allows the packets of a single flow to be marked at multiple ingress platforms. The definition can have strong security implications: any new packets added to the flow must only be added with the (explicit or implicit) agreement of the user. Further, the changes allowed for packets belonging to this flow are prescribed: in the example above, payload changes are disallowed. On-path platforms can easily change packets in any possible way, but the CHANGE platforms will not treat these packets as part of the original flow. It is up to the user to decide on the security it wants for its flow, and choose an appropriate identifier. In this process, the user will need to ask itself questions such as: what parts of the packet should stay intact? who can inject new packets in this flow?

In practice, multiple users may wish to run processing on the same packets (e.g. the destination ISP and the destination address) and it is difficult to re-use an existing flow definition; even if the users tag the same packets on the same ingress platform, they still need to "discover" their filters are the same, which could raise confidentiality concerns. Hence, it should be possible to add an arbitrary number of globally unique identifiers to each packet. Finally, users must be able to modify or remove packet tags belonging to their flows. This will allow users to split and merge flows, etc.

Implementing this ideal solution is challenging in practice for multiple reasons. First, the solution requires that each packet carries its own identifier(s), and protocol-agnostic implementations would necessarily be using IP options. However, IP options do not get through routers, so the solution cannot be deployed today. Other issues come from the cost of public-key operations which makes it difficult to achieve good performance.

The CHANGE solution sidesteps these two issues. First, we observe that only those platforms that process a flow need to identify packets belonging to that flow; this removes the need for explicit flow identifiers in each packet. Secondly, there is an existing trust relationship between the user and the platforms doing the processing. We leverage this relationship to remove the costly public-key operations from the data path.

At a high level, CHANGE uses a series of packet filters on each platform that needs to process a specified flow. In the CHANGE configuration file the user defines a new flow by giving a name, a set of packets that belong to that flow and a desired invariant for the flow. The packets are specified in the following ways:

- A filter (a bitmask) that is applied to incoming packets. This definition will be typically used at the ingress platform(s) (i.e. the first platforms that attract the packets).
- The output of a processing module.

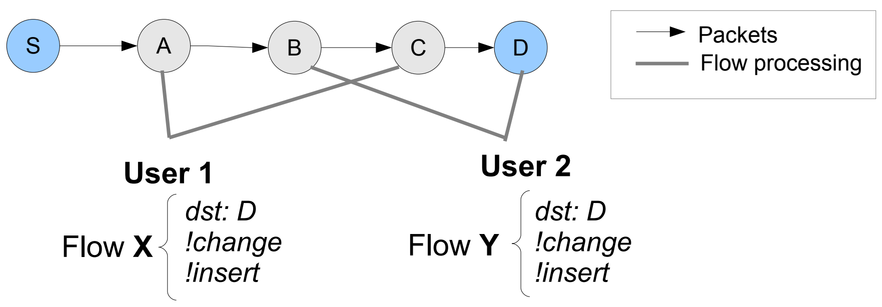


Figure 3.1: Flow naming problems when multiple users wish to process the same packets: user 2 will not be able to process the packets it wants on platform B.

- A combination of the above two.

The invariant specifies how/whether packets are allowed to be changed, or whether new packets are allowed to be injected by external parties after the flow has been created. If no invariant is specified, packet injection is permitted and arbitrary packet changes are allowed as long as the packet source and destination addresses are not allowed.

Once a flow is created, the user can use the flow name on different platforms to refer to the same packets. The platform transparently creates filters that match the packets on the destination platform as well as invariants as requested by the user. The filters take into account any flow manipulation requested by the user as well as the packet changes imposed by the invariants.

Note that the CHANGE architecture uses *implicit* flow definitions: these cannot be seen by observing the packets. Only the CHANGE user and the platforms involved in the processing are aware of the flow definition. Compared to the explicit identifiers we have identified as the ideal solution, this solution has similar functionality from the perspective of a single user.

However, when multiple non-cooperating users are interested in processing the same packets (such as the destination ISP and the destination host), the transformations one user makes to the packets of its flow can affect the packets seen by the other user. An example is given in Figure 3.1. Here two users wish to process the same flow—packets destined to D—on two platforms each. User X will select packets on platform A and these will likely be tunneled encrypted to platform C to ensure that the invariant specified—no packet changes, no new packets added—is satisfied. In this case, user Y will not be able to match the packets on platform B as it wishes. This problem is inherent in any implicit flow naming solution, but we have to accept it if we want deployment.

3.4 Types of Processing Primitives

We need to be able to reason about the security requirements and emergent behavior of flow processing functionality. To this end, it is useful to identify a small number of classes of flow processing:

- Read (R). The capability to read the contents of the packets to perform some action. Read-only, by itself, will not affect the flow.
- Filter (F). The capability to drop some or all of the packets of a flow, or to rate-limit that flow. Filter can change the behavior of flows, but not the contents of their packets.
- ReRoute (RR). The capability to change the path of a flow, but not otherwise affect the flow from an end-to-end point of view.
- Redirect (RD). The capability to change the destination of a flow.
- Modify (M). The capability to modify the header or payload of packets in a way that changes the end-to-end semantics of the protocols involved; this does not include changes to the IP addresses in the packets.
- Originate (O). The capability to originate new packets on behalf of another host.

These categories are fairly broad, and each can be instantiated in more than one way. However, they correspond to different security requirements, and having this classification allows us to reason about many (although not all) concatenations of flow processing. Actual instantiations of flow processing will often involve concatenations of these classes within the same platform or across platforms.

ReRoute and Redirect are the only classes that can change the path of a flow. Examples of re-routing include tunneling from one platform to another, or re-writing the source address on packets to be that of the processing platform.

Consider a bi-directional flow such as a TCP connection: tunneling of packets sent by the active opener pins the forward path to traverse the platform at the tunnel exit point. By itself this would not affect the reverse flow. A platform that NATs packets sent by the active opener would have the effect of pinning the reverse path to traverse the NAT. This also falls within the ReRoute class. Any platform that requires observing both the forward and reverse directions of a flow needs to pin both the forward and reverse paths, either using ReRouting or by virtue of being at a natural choke point in the topology that cannot be bypassed. A NAT is not the only way to pin the reverse path: a TCP-level proxy that split a connection into two could have the same effect.

The key differentiator of ReRoute from Redirect processing is that after passing through the flow processor, the flow continues on to its original destination and that the semantics of the flow are unaffected. The TCP-level proxy mentioned above is of course something of an edge-case. By using its own IP address for the outgoing connection, it clearly re-routes the reverse path. But the TCP packet headers and options are not preserved across the proxy. From an L4 point of view, it modifies the flow, but from an application level point of view it does not. In effect it performs ReRoute processing, but also acts as an L4 modifier. For classification purposes, we do not consider a NAT to modify a flow, though it does change the flow label, and this must be taken into account when establishing flow processing.

Another interesting example is an IPsec tunnel. This pins the forward path (and often the reverse path, depending on the deployment scenario), but it also pins the contents so they cannot be accessed or modified by additional flow processing modules between the tunnel endpoints. This capability to pin contents is a key attribute of some ReRoute processors, but we do not consider this to be a flow Modifier, as the transform is reversed at the tunnel endpoint, and the flow continues on to its original destination.

The Modify class is the most difficult to reason about. Flows can be modified in arbitrary ways so long as the path is unchanged - that is reserved for ReRoute and Redirect processing. Clearly we cannot provide a general framework to reason about arbitrary composition of arbitrary processing; however, if we specify *invariants* that restrict the type of changes that can take place, we can understand the emergent behavior. We describe such invariants in Section 3.5.1.

These classes of primitives offer a considerable amount of flexibility. Table 3.1 shows a broad set of processing functions and the way they fit into these classes.

Type	R	F	RR	RD	M	O
DPI	✓					
NAT	✓		✓			
Rate limiter	✓	✓				
Firewall	✓	✓				
IDS	✓					
IPS	✓	✓				
Transcoder	✓				✓	
Multimedia mixer	✓	✓				✓
Implicit Proxy	✓		✓			
Explicit Proxy	✓			✓		
Scrubber	✓	✓			✓	
Tunnel	✓		✓			
Multicast	✓			✓		✓

Table 3.1: Flow processing applications built from the classes of primitives.

3.5 Predictable Networking

To do useful work using the Internet, network users—such as end-users, applications running on endpoints or applications running in the network—expect a well defined behaviour from the network, and implement their functionality accordingly. This contract is fundamental: it separates the functionality implemented by both the network and the user, and ensures everything works as expected. It ensures transparency and allows users to easily troubleshoot network-related problems.

The original contract of the Internet was quite simple: the endpoints give an IP packet to the network, and the network does its best to get the packet to the destination. There are no hard delivery guarantees: the packet either arrives or it doesn't; if it does, the packet will be unchanged (from IP header up) or will contain one or more random bit errors. Packet ordering is also not guaranteed: packets sent in order by the source host may arrive out of order at the destination host.

This contract is easy to understand. To implement reliable delivery the endpoints need to retransmit lost packets, guard against reordering and check for errors in packets: all these can be easily accomplished by adding sequence numbers to packets (for reordering and loss detection), acknowledgments (to drive retransmissions) and checksums. Indeed, this is a subset of the features of TCP - the prominent transport protocol in the Internet.

The problem with the Internet today is that the contract between the user and the network has changed and in many cases the user does not really know what the new contract is. The minimum service level for home users seems to be "functioning" basic applications such as web browsing, email, VOIP, file downloads, etc: the network allows this type of traffic to go through, and we have no idea what happens to other traffic that looks "different". The minimum level of service offered by the network to mobile users is generally more constrained: here, web browsing and email typically work across the field, but not much more. VOIP is regularly blocked, TCP traffic is sometimes allowed only on port 80, and HTTP requests must be well formed; effectively the contract has moved from offering IP to offering HTTP for mobile users.

These observations are based on empirical evidence rather than something set in stone: the services offered (or rather allowed) by the network vary wildly between operators in the same areas, and around the globe. Our measurements presented in chapter 2 quantify the extent to this variability. The reality is that a typical user has no idea what network service it is getting: contracts do not regularly spell out such technical details. Even if they did, middleboxes exist at different places in the Internet, and users only have contracts with the first hop: they can't know what happens beyond that.

CHANGE opens middleboxes for innovation, allowing users and providers to instantiate new functionality that changes the network behaviour. To be able to implement new applications, CHANGE users need to know what the contract offered by the network is and understand whether their application will function correctly in this environment. This task is quite difficult:

- Today, the network contract is vague at least, and can change without notice.
- Tomorrow, CHANGE processing instantiated by other users or network elements will **dynamically change** the contract between the user and the network on timescales much shorter than today. Our security rules prevent other parties to enable remote processing unless they own the flow, but this would still allow a user's provider (that owns the prefix) to instantiate processing unbeknown to the user.

To mitigate the lack of information and allow predictable, evolvable networking, CHANGE uses *invariants*. *Invariants* can be used by endpoints to check or assert that certain properties they expect from the network service are indeed met. Invariants are used when the users do not know what the contract offered by the network is, or when they only have partial information from the network. In an ideal world, processing done by middleboxes should be *visible* to the traffic owners; this is not the case today, and it is unlikely to become so in the near future.

Invariants help deployment in an Internet where most middleboxes do not implement visibility mechanisms. The downside of invariants are their associated overheads (which vary depending on invariant type, as we shall see); if flow processing were visible to the traffic owners then it would be possible to statically check whether certain compositions make sense. We discuss such compositions in Section 3.5.2.

3.5.1 Invariants: Dynamic Contracts with the Network

In the general case, a user wanting to deploy new functionality does not know what service it will receive from the network: will the TCP sequence numbers be changed? will the TCP payload be changed?

A first observation is that the user may not care about most of the processing applied to its flows, as long it does not interfere with its own application. A great example of this is today: most users are fine with the Internet being heavily "optimized" for TCP and HTTP. Some applications or protocols, however, care that the TCP payload is not changed (for instance TCPCrypt or Multipath TCP).

Our solution is to have users express *invariants* about parts of the packets that should not be processed (changed or read) across a segment of the path (or the whole path). As long as these invariants are satisfied, the user is sure that his processing is behaving correctly, *regardless* of any other processing applied to his flows on that path segment.

The network's role is to *check* and possibly *enforce* these invariants. This will be accomplished by instantiating new flow processing functions at CHANGE platforms residing at both ends of the designated path segment. The processing may insert additional signaling (out of band) or even change the packets (by adding in-band signaling) to check the invariant.

Consider the simple end-to-end example we discussed earlier: where the IDS running on a CHANGE platform wants to make sure that the TCP payload is not altered on the path segment between it and the destination. To check that this invariant holds true, the platform can choose one of the following invariant implementation alternatives:

- Do nothing to check or enforce the invariant if it already knows the invariant is true. This would be the case when the path segment sits entirely within the AS of the platform's operator and no middleboxes are deployed that could change the flow, or when the path segment is already part of an IPSEC tunnel.
- Exchange periodically cryptographic summaries of the payload (such as a keyed HMAC [2]) of all the TCP connections. Changes would be detected, but after some time. This solution has very small overhead.
- Insert a keyed HMAC of the payload in each segment. Payload changes would be detected immediately, but the overhead is bigger—space is wasted in each packet.
- Encrypt the payload: changes to the payload are detected instantaneously as before. With symmetric encryption there is no space overhead as with HMACs. We observe that encryption offers a stronger

[2]the platform would first need to establish a secret key, which is easily done using the destination's public key.

invariant than just preventing payload changes; it also prevents anyone from reading payload contents.

The original Internet architecture offered a very strong invariant: packet contents beyond the IP header will arrive unmodified at the destination (or with random bit errors). Today's Internet holds invariant the TCP payload (for the most part), or preserves the HTTP semantics (via web proxies).

Invariants are dynamic contracts between the applications and the network. Useful invariants include: TCP payload, UDP payload, TCP ports, TCP options, TCP segment boundaries, TCP sequence numbers, etc. Invariants dynamically separate the tasks of the host and those of the network, building a common ground where application requirements meet network functionality.

Invariants leverage the reality that the network is better suited to implement certain functionality than the endpoints. Each operator knows what processing it applies to flows within its own network; it can then check that invariants hold statically, without adding any runtime overhead.

Although there are many possible ways in which all of these invariants could be enforced, full-blown IPSEC tunneling (or TLS, or something like TCPCrypt [11]) can not only check but also enforce most invariants. Why should we not use tunneling to enforce all invariants then?

The answer is efficiency: if the same end-to-end flow is subject to meeting different invariants by different stakeholders, it could easily end-up being tunneled more than once on the same path segment, adding useless overhead for no real benefit. In the long run, having several layers of encapsulation is a race to the bottom, leading to an evolvable but very inefficient Internet.

It is therefore important that the network implementation of invariants is as *efficient* as possible, being adapted to the characteristics of the path being monitored. If the same path is being monitored actively already, satisfying new invariants should *reuse* whenever possible the existing running functionality on the same path segment.

The job of the applications is to select proper invariants and inform the network. Applications might be tempted to just use the original Internet invariants, as they guarantee packets arrive unmodified at the other endpoint. The problem with this approach is that to enforce such strong guarantees it may be necessary to use IP over TLS encapsulation: this may get packets through but performance will be miserable when the network is even slightly congested (due to the tcp-over-tcp encapsulation problems). As they strive for efficiency and high performance, this gives applications strong incentives to choose the weakest invariant that still guarantees correctness.

Checking invariants is almost always possible, e.g. by using keyed HMACs. When apps discover their invariant does not hold true (say the TCP payload is changed), they can request the platforms to enforce the invariant. This will be accomplished with some sort of encryption, for instance with IPSEC. If the IPSEC tunnel is allowed through, then all is fine; otherwise a UDP encrypted tunnel might be created, and eventually a TCP/TLS one. If all of these fail, enforcing the invariant has failed - the app can now choose to give up, or try to find a weaker invariant that can be successfully implemented in the network.

A valid question is whether apps can use invariants to bypass operator policies. To some extent, they can as in the example above via encryption. However, operators can just disable all means of enforcing the invariants if they wish, blocking the traffic that does not comply.

It is important to note that applications themselves could enforce invariants if they wanted (and some already do). However, this would be **less efficient**: applications will enforce invariants for the whole end-to-end path, whereas network-based implementations can only enforce them on the path segments that would violate the invariants.

Invariants are not a panacea. Certain desirable behaviour is difficult to capture exactly with invariants, resulting in stronger invariants being requested by applications, which will lead to inefficiency. If every application wanted an invariant to be forced, this would result in an overall inefficient network.

The hope is that when enough applications request certain invariants the networks will just adapt their equipment to offer that invariant natively rather than as an added patch; this will also create healthy competition between operators.

3.5.1.1 Implementing Invariants

Applications provide the following information to processing platforms describing invariants:

- A list of processing primitives that *SHOULD NOT* be allowed to be executed on the given flow.
- A path segment where the invariant applies.
- An instruction to either check or enforce the invariant.
- A callback in case the invariant fails or cannot be enforced.

The processing primitives can be any from the list we have described in Section 3.4. *Modify* processing will also include the parts of the packet that should not be modified.

The path segment will be given as a pair of IP addresses. The addresses can be those of the endpoints of the flow, or they can belong to CHANGE platforms. The platform receiving the request will contact path segment's endpoints to setup the invariant. The platform will select the best of its invariant implementation strategies considering the user's preferences. If the invariant cannot be setup, the user will be informed at a predefined callback point.

Let's consider the simple DPI example from before, where the destination expects the TCP payload to be invariant from the DPI box to itself. For this it will create an invariant preventing Modify in the TCP payload on the path between the DPI box to itself, asking that the invariant is only checked to ensure minimum overhead. If the destination is notified that the invariant is not true, then it can reissue the invariant asking for it to be enforced.

To implement such invariants CHANGE uses its basic building blocks: platforms will be discovered using the CHANGE discovery service, then authentication will be performed to only allow traffic owners to request invariant processing. The security rules will be applied, and their net effect is that both source and destination can check invariants, but only the destination can enforce them (otherwise the destination has to explicitly agree to receive the flow). Once everything is setup, traffic can be redirected to the processing platforms.

3.5.2 Understanding the Composition of Flow Processing Functions

Let's assume all CHANGE platforms are willing to inform authorized users when queried about the processing being applied to their flows. The users can then utilize this information to decide whether their intended processing would work correctly if deployed; if the answer is negative, this same information allows users to select the best way to support the new functionality within the current contract offered by the network. The question we want to answer can be broadly stated as:

Is the processing instantiated by all parties (interested in a flow) working as those parties intended?

A simple example is this: if a filter aims to drop packets based on their source address, but the source address has been NATted, then the filter is failing its purpose. Can we detect such cases automatically? Is it possible to give a general answer to our question?

In this section we only consider the case where one party in the network knows about all the relevant processing instantiated (and maybe other relevant network state, e.g., routing tables). The information is collected from CHANGE platforms on the path of the flow to be processed. Given this information, how do we decide whether a certain set of processing functionality is safe to execute?

To make things more concrete, we consider that (combinations of) the following types of processing can be instantiated by different parties on the same end-to-end flow: Read (R), Filter (F), Redirect (RD), Reroute (RR), Modify (M), Originate. The original Internet can be modeled as follows:

$$O \rightarrow F \rightarrow R$$

The source Originates packets and gives them to the network. The network forwards the packets towards their destination, and can lose some of the packets if links are busy and buffers are full; this is represented as the F function in the middle of the flow. Compared to other Filter functions we will encounter, the Internet did not Read packet contents before deciding to drop them (hence an R operation does not precede F). Finally, the destination Reads the packet contents and performs useful processing.

Since virtually all applications can cope with lost packets (or TCP does for them), we observe that R can be performed regardless of filtering in the network. In the context of CHANGE, where R and F can be instantiated along the path, we derive the following rules:

RULE 1. Read functionality can be instantiated anywhere along a path and it will function correctly even if an unlimited number of Filters are in place.

RULE 2. Filter functionality can be safely instantiated anywhere along a path, as it will not impact the correctness of the applications Reading the packet contents (again, this is because by definition the Internet can lose packets anyways).

Note that Rule 2 just says processing will function correctly. It does not cover the following case where source S originates many packets for D that pass via T (D's intrusion detection system). S can mount a DoS attack and then drops all the attack traffic after T, hiding itself from the rest of the network and the destination:

$$S \to T \to F \to D$$

Reroute and redirect change the source and/or destination address of the packets, and the way they are routed. When applying Filter after RR/RD, Filter only works correctly if it is on the new path to the destination. Of course, Filter can be applied without restrictions before RR/RD.

RULE 3. Any primitive following RR/RD must be placed on the new path to function correctly. Mechanisms are necessary to detect the new path: if one party knew the state of the network—routing tables, link states, and so on—it could check whether rule 3 applies.

RULE 4. Read primitives following RR/RD are guaranteed to work correctly only if they do not depend on the source/destination addresses that were modified by RR/RD.

Current widespread use of NAT implies that neither endpoints nor network elements rely on knowing the true source address of the traffic (or at least they don't rely on it too much). Thus, it would seem that Read processing works fine after NAT-type Reroute. The same is not true for other re-routes - for instance RR could easily bypass destination-based filtering.

Originate can either create a new flow, or insert packets into an existing flow. In the first case, Originate is by definition the first processing function applied to that flow: nothing can precede it. When running other processing P after Originate, Rule 3 applies: P should be executed on the path to the destination to function correctly (this is true for any type of processing).

If we consider that Originate could be inserting packets in an existing flow, Rule 3 is no longer enough on its own: a Filter dropping all packets containing a certain signature can be bypassed by originating packets matching that same signature. To stop such insertion of packets downstream, Filtering could either be applied as the last processing before the destination, or it should enforce constraints to the rest of a path (such as no other packets can be injected). For instance, the Filter can also Modify the packet contents, either by encrypting them, or by adding a MAC that will be checked at or near the destination.

Reasoning about Modify processing is most difficult of all, but there are cases where we can tell whether processing is sage. One such example is when changes apply to different parts of the stack (i.e. packet headers), obeying the layering principles. For instance, processing that changes TCP headers but leaves the payload unchanged can be safely composed with processing that uses the payload only.

Understanding Arbitrary Compositions. As the rules above show, we can reason about many types of compositions ruling them as either safe or potentially unsafe. However, there are cases when two processing functions access the same part of the packet and their composition could or could not make sense, depending on the specific operations. Take the example of an MPTCP connection between two endpoints and an in-network middlebox on one of the flows:

- A traffic normalizer reads the whole packet (including payload), keeps per flow state and resends the same payload if it sees a new packet carrying an old sequence number it has already seen.
- An intrusion detection system reads the payload and checks to see if a virus signature is present.

The normalizer ensures that the subflow (which looks like a tcp connection) is consistent end-to-end, and works fine with MPTCP. The IDS on the other hand will fail since it does not see all the payload - half the bytes could go on the other subflow.

How can we automatically tell whether two potentially conflicting processing functions are safe to compose? To tackle this question we need:

(i) A way to succinctly describe the important functionality of a processing function. This could be achieved with a description in a high level language or just the implementation of that functionality.

(ii) A way to automatically test whether two functions can be composed.

One system that provides such functionality is Anteater [39]. Anteater translates desirable invariants into boolean satisfiability problems and checks them using SAT solvers against network state (routing tables, router configurations). Anteater offers a subset of the functionality that's needed by CHANGE , but it is a good start.

The CHANGE architecture does not prescribe a-priori an algorithm to check composition; today we could use the rules we've identified as well as tools like Anteater. We stress that checking composition safety in this way is inherently *centralized*, being run by one entity that holds enough information about the network.

To aid transparency, the CHANGE architecture mandates that visibility functionality must be built in all platforms. Essentially, each platform must be able to answer user queries such as: what processing are you currently applying to my flows? Supporting these queries are a series of architectural mechanisms including flow names (e.g., source or destination IP), user authentication, user-to-platform signaling and platform discovery.

We further note that, even if all middleboxes spoke CHANGE , there are still good reasons to hide processing from the endpoints: legal-intercept is an example where a tee is applied to the traffic originating from or destined to a monitored user, while the user should not know it is being monitored. It follows that, even in the best possible case where CHANGE is ubiquitous, *the user may not have information to decide whether it is safe to deploy certain functionalities in the network*. Thus, invariants offer a more complete solution.

3.6 Signaling

CHANGE uses four distinct types of signaling (these are described in great detail in Deliverable 4.2):

(i) **Discovery signaling** is used to find nearby platforms to one IP, and has three main components. First, it can be performed by the user using DNS to find the CHANGE anycast addresses; the user can then ping all of these and select the closest ones. Second, both platforms and users can employ on-path signaling to discover platforms (using techniques similar to traceroute). Finally, the user can contact a platform directly requesting it finds a platform nearby some IP; the platform could use a geolocation database and a list of available platforms to build a response (the latter is an instance of user-to-platform signaling).

(ii) **User-to-platform signaling** is used to carry requests from users to platforms and responses on the way back. Many types of requests could be serviced including discovery, setup of processing functionality (configuration files would be needed for the different elements). These requests need to carry flow definitions, user credentials and optionally billing information. We expect (but not mandate) that users will mainly contact their ISPs platform, and the latter will issue requests on behalf of the user to other platforms. This allows first hop ISPs to apply local policy to user requests, as well as proxy their requests if flow identifications change (as they do with a Carrier Grade NAT). Any request-response protocol could be used for this purpose; a good candidate is the web services framework, already used by major datacenter operators such as Amazon.

(iii) **Platform-to-platform signaling** is used to setup functionality involving more than one platform which includes setting up monitoring, enabling checksumming and various types of tunneling for inter-platform path segments. The signaling protocols must be able to include user and platform credentials as well the the required service. The requirements here also imply that the functionality should only be instantiated if all parties agree and have the necessary resources. For these purposes we propose the use of an inter-platform signaling protocol similar to RSVP-TE (as defined in Deliverable 4.2).

(iv) **Call-back signaling** is used by platforms to inform CHANGE users that certain events have happened. Example events include the request to setup a new flow using the user's IP as a destination, and the notification that an invariant has failed. Call-back signaling will be performed using a proxy based architecture (conceptually similar to SIP, but simpler): users register with their provider's platform and specify how they can be contacted (for instance they could leave a TCP connection open to the platform, in case they are behind a NAT). When an event has to be delivered to a user, the platform must be able to find the platform of that user, so that they can contact him; for this CHANGE can use mechanisms similar to SIP registrations.

3.7 Implementation Considerations

Auditing and Billing. Processing requests are accompanied by a cryptographic proof that the requesting entity either owns the subject flow or has been delegated by the flow owner to instantiate processing. Platforms maintain logs of all such requests, so that they may be consulted offline for detecting configuration problems, security auditing and billing.

Billing is seemingly orthogonal to the architecture, but it does depend on the deployment model. The deployment model further influences how the CHANGE architecture is instantiated, and what technical solutions are most appropriate.

We foresee two basic deployment models; in the first, a single entity, for instance a Content Distribution Network, deploys a global infrastructure that offers flow processing functionality. In this case flow processing users get an account with these CDNs, use an interface to make flow processing requests and pay the CDN for the services they use. In the CDN deployment authentication problems for on-path platforms are reduced, as there is an implicit trust relationship between all platforms operated by the same provider. However, flow processing is significantly more powerful then existing CDNs. Any one CDN deployment will likely be limited in the functionality it offers.

The second model is that of the Internet where a multitude of Internet Service Provides collectively provide global services. This is the deployment model we think is likely in the long term. The billing for such deployments is significantly more complex than the current cash flow in the Internet, with money flowing up the AS hierarchy. Solutions such as anonymous e-cash [18] seem more appropriate in this context.

Platform Discovery. An important part of the architecture is the ability to find appropriate processing platforms. Existing solutions to this problem exist in the CDN world, where client requests are directed to the closest server, typically by using geographical distance as a metric; these can be directly used by CHANGE . We also propose a novel solution based on BGP anycast that can offer better accuracy than IP geolocation, but is more difficult to deploy. A detailed overview of the different solutions is provided in Deliverable 4.3.

Traffic Attraction. Once an off-path platform is found, traffic needs to be drawn into that platform. Solutions here span a wide range, from DNS to BGP. We review these solutions in detail in Deliverable 4.3.

4 Security

An on-path platform will regularly receive flow processing requests from third parties. Accepting all requests creates obvious security concerns, and risks making the Internet even harder to understand and secure. In this section we describe what security rules are enforced in the CHANGE architecture, we give the reasons for choosing these rules among the many possible in section 4.1.

We use two layers of defense against potential misuse. The first layer restricts the users that can request processing: CHANGE platforms allow processing to be instantiated only by traffic sources and destinations. Authenticating the traffic source and destination is relatively straightforward; in principle, we can check that the requesting entity is the rightful owner of the source or destination address of the IP packets. If the request involves a prefix rather than a single IP, authentication could be performed at the prefix level.

There are two practical ways of implementing such authentication in today's IPv4 world; leveraging DNSSEC reverse look-ups or using the newly proposed RPKI infrastructure. The basic idea in both of them is that IP address owners will have a public/private key pair which they can use to authenticate themselves. The public key will be delivered with X.509 certificates authenticated either by the DNSSEC or RPKI infrastructure. We discuss these and other alternative solutions in greater detail in Deliverable 4.1. Because of the way address allocation works, both RPKI and DNSSEC support authentication at the prefix level.

Extending such authentication all the way down to the IP address level is a matter of deploying RPKI/DNSSEC infrastructure at every ISP; the resulting host-based certificates could be deployed via new DHCP options (as proposed in D4.1). Alternatively, an ISP may just sign processing requests from hosts with its prefix's secret key. The downside is that this allows L2 impersonation attacks within the ISP network, for instance ARP spoofing in 802.3 networks.

In IPv6, authenticating individual traffic sources or destinations is easier as it does not require PKI. The 128 bit IPv6 address is split in two equal parts: the network part of the address that is used by the routing system and the host part. Hosts can freely choose the host part of the address with Stateless Address Autoconfiguration. This allows hosts to create addresses that are self-certifying; a given host may create a public/private key pair, and use the first 64 bits of a cryptographic hash of the public key as the lower 64 bits of the address. The host can then prove to CHANGE platforms it owns the address by using its secret key. This is the concept used by Cryptographically Generated Addresses (CGAs) [4]. CGA's do not allow the ISP to authenticate itself; to support it, RPKI is also needed.

In certain cases direct authorization is not possible nor desirable. Consider a destination wishing to push out filtering against a DDoS attack: the destination will want to delegate nearby platforms to instantiate further filtering upstream, with the aim of stopping attack traffic close to its sources. To allow such functionality, we employ (as in previous work [59]) capabilities that allow a third party to perform restricted types of processing on behalf of the owner of the address, for a predefined period of time. In the remainder of this document we refer to the owner or the delegated owner of an address as the delegated owner. Delegation requires that endpoints provide the delegated platform with signed certificates to perform the listed action. They are valid only for the duration of the delegated operation.

Our second layer of defense is a set of rules that specify when platforms can instantiate processing on behalf of other parties, and what type of processing is allowed. These rules are enforced by platforms and their net effect is the prevention of in-network spoofing as well as providing a default-off, destination-based architecture for re-routed flows.

Only traffic sources and destinations are allowed to request processing regardless of the processing involved. Within this envelope the architecture must enforce the following low-level security rules:

(i) **Changing Source Addresses** is permitted only if the new address has been delegated to the requester [1].

(ii) **Changing Destination Addresses** is permitted only if the new address has been delegated for use by the requesting party. The net effect is a default-off behavior when traffic is re-routed from its destination-based path or redirected to a new destination: unless the (new) destination agrees, the operation will not be allowed.

(iii) **Implicit authorization**. When a host A initiates a new connection that is visible to the processing platform and destined to the platform or one of its clients, the platform is implicitly authorized to use A as the destination address of the response traffic. This rule is similar in spirit to existing firewall behaviour that allows incoming traffic corresponding to host-established outgoing connections. This rule directly allows the traffic destination to instantiate a web server as a processing function.

These easy-to-enforce low level rules have direct implications for authorizing processing that re-routes, redirects or originates traffic.

Pinning the forward path (e.g., via a tunnel) is directly allowed for the destination of the traffic: it will contact the tunnel entry point proving it is the destination of the traffic and will request that traffic is tunneled to an exit point. The tunnel endpoints will communicate to verify their credentials, and to communicate the destination's request. Encapsulation is allowed because both source and destination addresses belong to the platforms. Decapsulation is allowed because the destination authorizes the destination address rewrite at the tunnel exit point.

The source of the traffic cannot pin the forward path: it is not allowed the decapsulate step at the tunnel exit point. This must be authorized explicitly by the destination. Although some flexibility is lost, the fact that the destination has route visibility helps ensure the correctness of flow processing.

Both the source of traffic (active open) and the destination can pin the reverse path by NATing. For NATs we make the further restriction that the platform must use its own IP address as the new source address of the traffic - this helps the accountability of the network. On the return path the destination address has to be

[1] When the platform owns the new address it will temporarily delegate the address/port to the requester

rewritten to match the source's IP; this is directly authorized when the source requests the NAT (the source owns the address) and indirectly authorized when the destination requests the NAT (because the source has initiated the connection).

It follows from the above that only the destination (or passive opener of a connection) can pin both the forward and the return path of a connection.

The security rules restrict originating packets from a platform to require the authorization of the destination. The destination rule allows implementing multicast functionality as follows: the source first instantiates an inactive "tee" functionality at certain platforms in the Internet. Receivers connect to these platforms specifying the identifier of the "tee" and authorizing the platforms to send traffic back. From this point on, traffic flows from the source to the destinations, being "split" at the platforms.

The security rules do not directly influence read, filter and modify processing functions: both sources and destinations are allowed to instantiate them on path. However, stakeholders can restrict the set of processing other clients can run on the same traffic, as we discuss next.

Consider implementing a Content-Distribution Network with a flow-processing platform: the web server authorizes the platform to attract packets destined to it (either via DNS or BGP FlowSpec) and stores the relevant content. When a web client makes a request, the CDN will receive it and process it; however the CDN (acting on behalf of the web server) is not allowed to send the packets back, unless the web client explicitly authorizes it (e.g. by delegating the platform).

The solution is provided by Rule 4 which implicitly authorizes destination address rewrite on the return path, on the basis that the web client has created the request in the first place. This is similar in spirit to firewalls allowing outgoing connections and only their corresponding return packets.

4.1 Design Choices

The CHANGE architecture envisions high performance, flexible flow processing. However, if all users are allowed to freely instantiate processing or retrieve statistics for all other flows in the Internet, CHANGE can be easily be misused. In this section we describe the design space related to security and explain our decisions on securing the CHANGE architecture.

The first important question is who may request flow processing? For any point-to-point message exchange, a reasonable answer must be centered on the "owners" of the flow and could include the traffic source, the destination, or the network boxes that see the traffic. (We can generalize this definition for many-to-one and one-to-many traffic patterns by limiting each user to process packets coming to it). Should we allow all of these to request processing?

It is obvious that the end-hosts or any network element "owning" a flow (i.e. observing the flow's packets going through) can today apply any processing it wishes to those packets. CHANGE cannot really stop all such flow processing from happening, as on path platforms can always drop or redirect/reroute packets. However, CHANGE can check and enforce invariants, restricting the processing that is permitted to on-

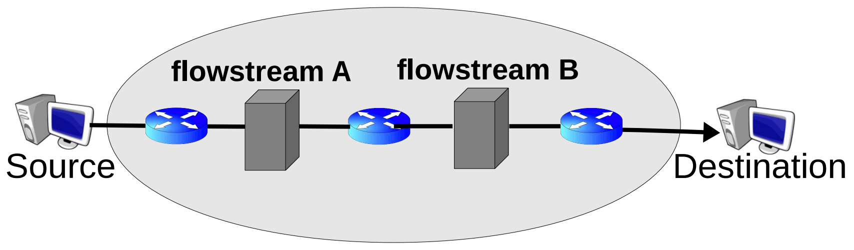


Figure 4.1: An end-to-end flow in the CHANGE architecture

path platforms to a smaller subset—for instance, encryption dissallows payload read and payload modify operations.

We stress that the question is: out of the entities that own a flow, which are allowed to request processing to be performed *on their behalf* from other platforms? The answer is not straightforward, as it also depends on what type of processing is in question. Traffic monitoring is essentially a read-only operation, and bears less security implications than, say, rewriting packet contents.

We note that there is an interesting parallel here to the roles of entities in the routing system today: traffic (or flow) ownership is centered on the destination; hosts (or networks) can choose to announce or withdraw routes to their prefixes depending on whether they wish to be reachable or unreachable. Multi-homed sites can choose one or a subset of uplinks on which to announce their addresses, intermediary routers rank routes to the same prefix according to their own preference, in order to influence the paths of the traffic that passes through them. While the source should be able to control the path its traffic takes, in practice it has the least control over its packets, because extensions such as Loose Source Routing are not widely supported [24].

4.1.1 Once You Forward the Traffic, It's Yours

Particular properties of the current Internet architecture constrain the possible solutions to flow and entity authentication. It is easiest to see this with the example shown in Figure 4.1. First, assume that only the `Source`, as identified by the IP source address in the packet header, is allowed to make processing requests. Even if we assume that the source can cryptographically prove that it owns the address it refers to, any on-path platform can network-address translate or even tunnel packets it wishes to have processed, thus pretending to be the source. In our example, flowstream `A` can NAT the flow and request processing on flowstream `B`, pretending it is the source of the traffic. In the current Internet this is perfectly acceptable, and there is no way to prohibit such behaviour. The implication is that *any on-path entity has full downstream control* of the flows it forwards *if source authentication is allowed.*

Now assume that only the destination (`DST`) of a flow is allowed to request *any* flow processing functionality, including changing the destination address to a new arbutrary value. As illustrated in Figure 4.2, consider an on-path platform `A` that sees the traffic to the destination and wishes to have it processed downstream by another platform `B`. Platform `A` can apply the following strategy to convince `B` to do processing on its behalf: first `A` creates a tunnel from itself to `B`. Then, `A` takes each packet destined to `DST`, rewrites the destination address to its own address and sends it through the tunnel to `B`, and finally at `B`, it decapsulates the traffic.

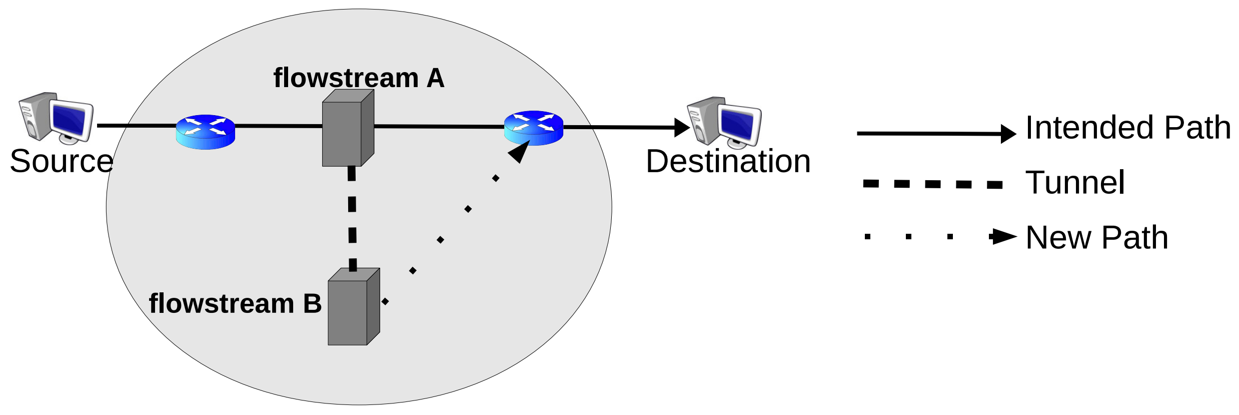


Figure 4.2: Being On-Path Enables Nodes To Re-direct Traffic Arbitrarily

Since B (which was previously off-path) now sees traffic exiting the tunnel labeled to be destined to A, it will consequently allow A to instantiate new processing on the flow. A can then rewrite the destination address of the packets back to DST, redirecting the traffic back to its original destination.

Downstream Flow Processing Property: if either the source or the destination of a flow are allowed to request in-network processing of it, *any* on-path network element can request processing from other *downstream* platforms.

Note that the *downstream* distinction is very important; it says that a platform must be able to *see* the traffic before it can request processing for it. In general, this effect seems benign: a platform that already has full control of the traffic can exert its control downstream.

If the traffic going through a platform is encrypted (e.g. an IPSEC tunnel), can a platform misuse flow processing to read traffic contents downstream? The answer is no: the platform can control downstream processing up to the tunnel exit point, which is the current destination of the flow. The platform has no control after the traffic is decapsulated.

Our solution: restricting unauthorized destination-address rewriting This straightforward solution mitigates both the security issues and simplifies the detection of reachability problems related to arbitrary destination address rewriting. All we need is to request that the entity wanting to rewrite the destination address should either own it or should be delegated to do so by the owner.

In Figure 4.2 this requirement would imply that S should ask permission from $\frown{D}$ if it wants to proxy traffic to D via B. If D grants the permission, it gives A a delegation certificate which B will use to authorize the proxy. This implies that on-path proxies cannot instantiate arbitrary downstream processing with primitives reserved to the destination. Reconsider the attack shown in Figure 4.2: A can request B to tunnel, but it cannot redirect the traffic coming out of the tunnel to DST; it is forced to route it back to one of its own addresses. In other words, downstream flow processing can be effectively forbidden from certain primitives if only the destination is authorized to use them.

Delegated authorization also introduces a few restrictions when the same primitives are used for legitimate purposes. Consider a source, such as a web server, wanting to create two distinct paths to a given destination; the host may run Multipath TCP [27] on top of these paths to get increased robustness and throughput for its flows. A strategy would be to choose the default path provided by BGP as the first path, and to proxy traffic via a way-point to create the second path. The trouble is that sources cannot instantiate proxied connections or even tunnels without the destination agreeing.

In the early days of deployment, few ISPs and fewer hosts will support flow processing. In this case there is no way that the source can get permission to proxy its traffic. The same restrictions apply to tunnels, another tool that is instrumental to the evolvability of the network. Such restrictions do limit the incentives to deploy CHANGE , reducing the competitive advantage of early adopters.

To conclude, we choose the more restrictive solution of having destinations explicitly agree with their address being used to redirect a flow: while this does limit the use-cases of CHANGE , it crucially ensures the infrastructure cannot be misused.

4.1.2 On-Path Platforms Should Not Be Allowed to Instantiate Upstream Processing

Safely authenticating on-path platforms is quite difficult as it requires in-band, continuous solutions; on-path challenge-response protocols run at specific points in time are not enough to guard against on-path attackers that move off-path after processing is started, while filtering precludes in-band authentication altogether. Such circumstances raise the question: should we allow on-path platforms to instantiate processing on upstream platforms for *any* of the listed primitives?

The primitives that seem to pose the least security threats are in the read category, and seem to be the best candidate.[2] It turns out that even in this case in-band solutions are either insecure or difficult to implement.

To understand why this is the case, we can think about two possible strategies for implementing them. In both cases, a platform is requesting upstream processing. The first strategy is to use on-path challenge-response when monitoring is started. The first step here is for the user to provide an IP bitmask specifying the source/and or the destination of the traffic to be monitored, and challenge-response would need to happen on each individual IP that matches this bitmap. However, checking one or a few individual flows is not enough to defend against attackers that see a subset of traffic. Hence, a majority of traffic needs to be included in the challenge-response phase; platforms that only pass a subset of the challenge response test would not be allowed to monitor traffic. This last solution breaks down if there is genuine packet loss; the verification platform cannot reliably check whether packets are lost or routed elsewhere, away from the requesting platform. Similar problems appear with in-band monitoring, we could address this in a number of ways, for instance by adding a total packet count to each prefix, but this would allow an on-path attacker both to estimate the total traffic volume going through the monitoring platform, an undesirable effect since such

[2]Full packet capture (a read operation) along with transfer of the information to a remote site is equivalent to a tee, and thus has further security implications. However, for the remainder of this discussion, let's focus our attention on requesting packet-level statistics (such as counts, drops, rates).

information is considered sensitive by ISPs.

We conclude that on-path platforms should not be allowed to instantiate upstream processing.

The platform must provide a way to delegate authorization from traffic endpoints to the on-path platforms. There are straightforward ways of implementing delegation: the endpoint can sign a delegation certificate that specifies the delegated platform's IP address, the amount of time the delegation is valid for and the types of processing primitives allowed. When making a flow processing request, the platform would authenticate its own address and present the delegation certificate. To ensure correctness, platforms and endpoints must be loosely time synchronized.

4.2 Security Properties

The design of the CHANGE architecture aims to strike a balance between flexibility and security. Adding flexibility to the ossified Internet is paramount, and we achieve this by allowing traffic sources and destinations to request and instantiate in-network processing. Avoiding abuse is done through a series of rules that platforms must enforce when instantiating processing on behalf of others; these were reviewed in Section 4.

DDoS Protection. Processing platforms cannot themselves be used to launch Denial of Service Attacks against endpoints because of the requirement to have explicit authorization (security rule (ii)). To defend against traffic originating from existing hosts or misbehaving platforms, destinations can instantiate traffic attraction and filtering primitives on remote platforms close to the traffic sources.

Routing Policy Enforcement. Destinations have ultimate control over the path taken by incoming packets (rule (ii)), allowing them to implement routing policy as needed using a mix of traffic attraction functions such as tunnels, NATs and routing announcements. The spirit of the CHANGE architecture is default-off: any packet created by CHANGE platforms must be explicitly accepted by the destination.

Preventing Spoofing. ReRoute, Redirect and Originate primitives do not provide spoofing opportunities as they require new source addresses to be owned by the entity requesting the processing (rule (i)). However, the Internet is not secure against spoofing and our architecture cannot remedy this. Having said that, if the owner of the source prefix supports flow processing, the destination can instantiate read-only filters to check that traffic indeed originates from that address.

Improved Accountability. Flow processing provides a way of discovering and monitoring processing instantiated by cooperating platforms and a way of detecting the existence and pinpointing the location of misbehaving middleboxes by monitoring flows at multiple vantage points en-route to the destination. Invariants allow traffic sources and destinations to explicitly say what service they expect from the network, enabling new extensions and making the network much easier to debug.

Security Limitations. We made the conscious choice of only allowing traffic sources and destinations to request processing, because authenticating a request from an on-path ISP opens up too many security holes. If a DDoS attack targets an ISP by sending to downstream destinations, our framework cannot help without the destination's consent. However, ISPs may issue requests using the same platforms to defend against such attacks if a pre-existing trust relationship exists between the relevant ISPs.

On-path platforms, however, can instantiate *downstream processing* at their will because they can NAT the traffic, effectively becoming the new source of the traffic.

5 Motivating Scenarios

To understand how CHANGE would work in the current Internet we select a few interesting scenarios that are difficult to implement today and we discuss them in greater detail.

Deploying New Transport Protocols is our first scenario and looks at how network invariants can help deploy transport protocols in an efficient way. This use-case is instrumental in empowering the users to evolve the network (i.e. deploy new applications).

Our second scenario, *Inbound Traffic Engineering.*, is an example application useful for ISPs that is very difficult to implement today with fine granularity. It shows that CHANGE is equally useful to both end-users and ISPs.

Our third scenario focuses on tackling *Distributed Denial of Service Attacks.* DDoS attacks have been a common occurrence for many years, and there are no signs of them going away anytime soon.

Finally, in the *Monitoring* scenario we show how CHANGE helps users understand what is going on with their traffic, and helps them both debug the network and their flow processing, as well as helping them to make the right choices in enabling the desired functionality.

5.1 Deploying New Transport Protocols

The Internet has ossified, and the net effect is that we are "locked in" to an IP world. Deploying new transport protocols faces many hurdles, the biggest being that packets with new protocol numbers don't get through the network.

In the short term, then, new transport protocols must be tunneled over existing protocols such as UDP and TCP. Tunneling over UDP is possible but UDP has a much harder time getting through the various middleboxes than TCP. Should we tunnel new transport protocols over TCP then? This would be nothing short of disaster, as the new transport protocol will inherit TCP's built-in functions such as reliable and in-order delivery, together with their associated problems such as head-of-line blocking.

What is needed is a way to create short TCP tunnels for the parts of the network that only allow TCP to pass through, and a way to "glue" these with segments that only need UDP encapsulation. Once there is partial support for a new protocol, native forwarding should be used for the segments of the network that support it.

Once we tunnel traffic over UDP or TCP, we inherit the "optimisations" embedded today in the Internet for this type of traffic. This means that the TCP and UDP packets sent might very well differ from those received. CHANGE allows participants to express invariants that the platforms will implement as efficiently as possible. These invariants will be used to ensure that the packet contents are not altered in a way that is detrimental to the new transport protocol.

CHANGE can be used to create these tunnels and glue them in a coherent end-to-end path whose properties are well understood. CHANGE hosts (source or destination nodes) may discover on-path platforms and setup tunnels between themselves to "hide" traffic from misbehaving middleboxes by creating an encrypted tunnel to ferry the traffic between intended targets. To instantiate the tunnel, the initiating host may contact the platform target of interest and setup the "payment" information. The receiving host does not need to authenticate the host to setup the tunnel; it will need to authenticate it if further processing is required, for instance network address translation. Note that implementations of this scenario need to take into account existing on-path NATs.

An issue of concern for tunneled traffic is the question of how the source address is defined after the traffic exits the tunnel. The problem here is that the source address of traffic may be used to define its flow id, for example as is the case for the 5-tuple flow id structure. To address this point, the source host may choose to change the source address of the traffic exiting the tunnel, as if it were running a NATing at the tunnel exit point. This is possible because the source host has full control over the source address of the traffic that entered the tunnel, so it is in a position to know the previous identifier information including the original source address. Such a set up would also force the return traffic to enter the tunnel, and can enable hosts to have a private (non-routable) IP address.

A second case is when the source host wishes to setup a tunnel for a path segment in the middle of the network. In this case, it needs to discover and contact the tunnel entry point and authenticate itself as the source of the traffic. The tunnel entry point then contacts the tunnel exit point and creates the tunnel. In this case, we are dependent on destination-based routing to direct the traffic to the tunnel entry point. If we want to ensure that traffic always goes through the tunnel, we have to either attract destination traffic to the platform or the source has to rewrite the destination address of the traffic to be the platform (preferred). Now the tunnel exit point must behave like a proxy, rewriting the destination address. If we also want the reverse traffic always cross the tunnel, NATing is again required at the tunnel exit point.

In practice we expect that endpoints will instantiate path segments when needed for the "access" part of the path, to bypass deployed middleboxes, as the core is currently just doing plain packet forwarding. Complexity for this step is reasonable, as in the worst case one network address translation per path segment needs to be setup. When the path being created is in the core, though, things are more complicated, as hosts may need to instantiate both a NAT and a proxy for both directions of traffic or to otherwise draw traffic into processing platforms, for instance via BGP.

5.2 Inbound Traffic Engineering

In this scenario, we consider a stub Autonomous System (such as a cable or DSL provider) that is multi-homed to different upstream providers and wishes to balance its traffic over all the links. A concrete example is AS 1 in Figure 5.1 that has 5 uplinks to upstream ASes. Currently ASes advertise different IP address blocks over their uplinks, but this is coarse-grained and increases the size of the global routing tables. In IPv6, only the network part of the address can be advertised, and this technique is effectively unusable.

With CHANGE , AS 1 can choose to advertise its prefix only on two of its uplinks, and it sets up tunnels from a CHANGE platform in AS10 to use its three links. Within AS 10, the CHANGE platform will attract

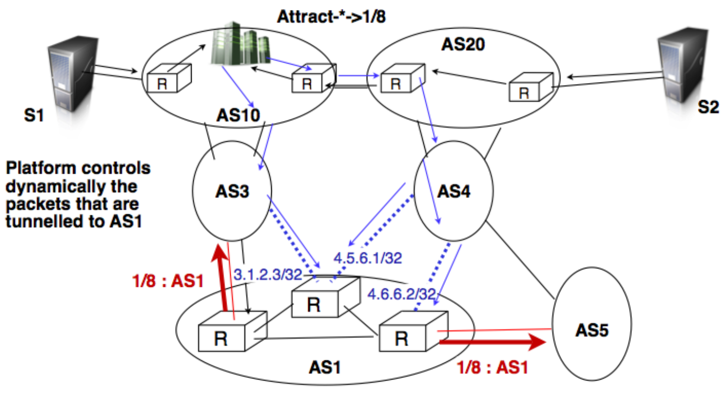


Figure 5.1: Stub AS Performing Inbound Traffic Engineering with CHANGE

all the traffic destined to 1/8 by using FlowSpec. AS 1 can fully control traffic reaching it via AS 10 using processing on the CHANGE platform: it can choose to receive it using destination-based forwarding or via one of the three tunnels.

AS 1 owns both 1/8 and the three tunnel exit points; it can use its RPKI certificates to get authorization for processing. The configuration will be more effective in load-balancing when the amount of traffic crossing AS 10 (the platform) bound for AS 1 is large. If this is not the case, the host can try to get more traffic into AS 10 or use another platform upstream of AS5.

Knowing the topology helps speed up the process of instantiating effective load-balancing. AS 1 can discover nearby platforms and check their AS numbers, and use information on the AS level topology to instantiate processing at the right place in network. However topology information is sensitive and not easily disclosed by ASes; if no topology information is available the stub AS can still use trial-and-error to look for an appropriate load-balancing configuration.

5.3 DDos Filtering

Instantiating DDoS filtering is a simple process with CHANGE . The destination can use its RPKI secret key to authenticate itself as the traffic owner (or will have its own provider sign its requests using the prefix's secret key).

If AS-level topology information is available, the destination needs to discover and instantiate processing at its closest CHANGE platforms (directly connected/upstream ASes), selectively dropping the malicious traffic. If the traffic volume is too much for the current set of platforms to handle, filtering must be pushed further upstream. The destination will delegate platforms to act on its behalf and request further filtering, quenching the traffic volume.

5.4 Monitoring

Today's Internet harbours a myriad of middleboxes that independently perform useful functionality. The behavior of existing protocols like STUN/ICE is difficult enough to predict on their own, but when combined with other middleboxes that change, drop, scrub packets, re-segment or pro-actively acknowledge TCP traffic, understanding network behaviour and finding the causes of problems is a daunting task.

CHANGE can help here with its “read-only” primitives that allow traffic owners to examine their flows at different vantage points in the network. Traffic sources and destinations can instantiate (preferably on-path) processing and directly compute packet loss rates for path segments, detect packet changes (e.g. dropped TCP options), and so forth.

6 Related Work

Despite all research efforts done in networking, and the enormous range and popularity of the applications that are running over the Internet, the core architecture of the deployed Internet infrastructure has not changed greatly in the last 15 years.

Many application-level solutions have appeared to answer the deployment barrier, but these solutions are network and operator policy agnostic, will therefore tend to "hurt" the network if used in large-scale. As highlighted, one of our purposes is to enhance the Internet architecture in a bottom-up manner, adding flexibility and new processing primitives in a way that provides benefits to early adopters, while ultimately being guided by the much greater benefits to be had from the coordinated services.

So what is the current network architecture and how can we bypass existing issues to obtain more flexible solution? We begin this chapter by briefly highlighting history of network architecture, and then provide a brief survey of more recent trends in research, pointing out how CHANGE stands out.

6.1 History of network architecture

In this section we present a brief history of network architecture, discussing three broad architecture classes and their relationship to CHANGE , these are; circuit and flow switching, programmable networks, and service-oriented architectures.

6.1.1 Circuit and Flow Switching

The Internet architecture as we know it today has been based on both the datagram model and the end-to-end principle. The datagram model assumes that all packets are treated independently in the network while the end-to-end principle puts intelligence only in the end-systems. These two assumptions have been challenged during the last two decades.

The first approach to support Quality of Service in the Internet, Integrated Services (Intserv) [13], included layer-4 flows in the network and required QoS state in the routers. Unfortunately, using layer-4 flows made the integrated services architecture unscalable and it is still not deployed today in the global Internet. Differentiated Services (Diffserv) [12] appeared later as a more scalable alternative to support QoS. Compared to integrated services, Diffserv introduces less state in the network as it deals only with highly aggregated flows. Diffserv is the basis for several uses of QoS in today's IP networks. Multi-Protocol Label Switching (MPLS) [46] appeared initially as a solution to better integrate layer-2 techniques such as Asynchronous Transfer Mode (ATM) and FrameRelay into IP networks. However, it quickly evolved as a new layer-2.5 technique with its own header below IP. MPLS is widely used by ISPs to provide services such as VPNs, VoIP or IPTV. It serves as the basic infrastructure to allow an operator to support multiple services on top of a single network. Furthermore, GMPLS [10] allows optical networks to be integrated in the IP architecture in a coherent manner. MPLS also deals with flows and adds state to the routers. Compared to DiffServ and integrated services, MPLS is slightly more flexible in the definition of the flows that it supports since several types of Forwarding Equivalence Classes can be defined. However, most deployments define a flow, either as all the packets that need to exit a network through an egress router, or as all the packets exchanged between an ingress and an egress router. From a QoS viewpoint, the main benefit of MPLS compared to Diffserv is the ability to forward flows along chosen paths to meet traffic engineering constraints.

In parallel with the evolution of support for QoS, we have more recently seen growth in the use of middle-boxes in IP networks to provide services that were not considered when the architecture was designed. These include deep-packet-inspection, shaping, firewalls and denial-of-service protection and new services are created. These services need to be able to process particular flows at particular locations in the network. Compared to existing flow-based techniques, an important benefit of CHANGE is that it will support a flexible definition of the flows to be processed. A second benefit is that it will be possible to instantiate some processing, whatever its complexity, inside the network on flow processing platforms. These flow processing platforms will either be on the path followed by the flows to be processed or the network configuration will be changed to route flows via these platforms. More recently, the OpenFlow [41] concept has been gaining momentum. OpenFlow switches are programmable via a so-called flow table that can be configured by adding entries. These entries aggregate packets into flows by matching on a number of L2, L3 and L4 fields and then specify the switch port to which the flow should be sent. In this way, OpenFlow provides a basic L2-L4 flow classification mechanism. CHANGE will use OpenFlow switches as a fast classifier in its flow processing platforms, but by also including more programmable hardware we will provide more advanced processing capabilities.

6.1.2 Programmable Networks

Active and programmable networks [56, 57] were two packet processing models proposed in the mid-1990s. They shared with CHANGE the laudable aim of enabling more flexible in-network processing. The major differences between the three lies in the processing model and implementation approach used. In active networks, data packets carried code to be executed in routers, while in the programmable network model the code could be pre-loaded in the routers. In both these proposals, the implementation model was that this code was typically executed on either custom router hardware or on commodity PC hardware adjunct to the router. As both computational models used packets as the processing unit and hardware speed was severely limited compared to todays capabilities, performance was a critically limiting scalability factor. Also, because the approach was packet-centric, the deployment models often required that most, if not all, routers in the network (and sometimes the network stack in end-hosts) had to be upgraded to achieve desired benefits. As a result, neither active nor programmable networks gained wide acceptance and both failed to see any deployment.

In contrast, CHANGE proposes the use of flexible flow definitions as processing units. This means that only the traffic of interest is handled by the programmable flow processing platform. The only operation that takes place at the packet granularity is classification, where CHANGE exploits, and greatly benefits from, high-speed, cheap and commoditized hardware solutions (e.g., TCAMs in routers and OpenFlow switches). On the processing front, CHANGE also benefits from high-performance multi-processor, multi-core systems. Implementation of the CHANGE processing model can be supported either by off-the-shelf high-performance custom components (e.g. network processors or tile processors) or even by current high-performance commodity server hardware (with their multi-core CPUs and high speed system interconnects). Finally, CHANGE assumes that flow processing platforms will be deployed only where needed, exploiting virtualization to improve sharing of the platform and isolation of processing.

The major advantage of such a deployment model is that all the assumptions and characteristics that made the existing Internet so successful are retained, while the notion of flows and processing are only actuated at processing points. For all these reasons, we therefore believe that CHANGE is poised to succeed where previous attempts at in-network processing failed, and see an incremental, but eventually wide, deployment. More recently, router programmability has seen a resurgence of interest.

Projects like Vrouter [23] or RouteBricks [2] exploit software router platforms to provide customized fast paths. Here, the unit of customization is the router itself so there is no attempt at providing processing differentiation on a per-flow or per-packet basis. When used in conjunction with virtualization techniques, these approaches allow the sharing of a common hardware box or cluster between several independent software routers, each running a custom network stack and belonging to a different virtual network. Super-Charging PlanetLab [53] has the same goal but uses network processors to implement the fast paths of the virtual routers. CHANGE builds upon this work, and will support flow-centric processing within the same hardware context as these programmable routers.

6.1.3 Service-oriented Architectures

Until recently, innovation in the Internet was mostly technological, fuelled by the ever increasing need for speed, growth and reach. This era saw rapid increase in network bandwidth through the advent of high-speed switch fabrics, optical networking and fast IP lookup, to name just a few. Access to the network was improved by “last mile” technologies such as ADSL or wireless access. As a result, innovation was mainly the realm of equipment manufacturers. However, changes to the architecture itself have met much resistance. For instance, in the mid 1990s there was a great deal of interest in IP Multicast as an enhancement to the basic Internet architecture that would support multi-point distribution. In practise, although almost all Internet routers now support multicast, it is rarely enabled inter-domain. In recent years with the rise of IPTV, multicast has once again become popular with consumer ISPs that wish to distribute video, but it tends to be limited to the edge ISPs. There are two main reasons for this. First, there is a chicken-and-egg problem: application writers cannot assume that multicast exists, so don’t add multicast support; ISPs can’t see any application demand, so don’t enable multicast. Second, IP multicast is complex and difficult to manage, except in limited deployments. There are real costs for deploying multicast, even though the hardware supports it. The effect of this failure to deploy multicast has been that application writers have implemented application-level workarounds. For example, the BBC developed iPlayer, which performed peer-to-peer distribution of high-quality video of all the BBC's programs. While the application worked well, the effect on British ISPs was predictable; they were overwhelmed with additional traffic taking paths which were suboptimal. In response ISPs started to implement deep-packet inspection and traffic shaping for iPlayer traffic.

It is just this sort of response that CHANGE hopes to avert. In-network flow processing platforms have the potential to serve as traffic fan-out points, but to do so in a manner that satisfies both the ISPs need to manage their network and the application's need to push the same content to many customers. It is unlikely that it is worthwhile deploying dedicated infrastructure for such a task (after all there is still a chicken-and-egg problem), but CHANGE allows this to be done in flexible flow-processing infrastructure installed for other purposes, and thus the up-front costs are small. In addition, CHANGE allows this fan-out to be done outside of the core IP forwarding path, avoiding the risk associated with deploying new services in critical infrastructure. The same arguments apply to other fundamental architectural changes: CHANGE lowers the costs and barriers to entry, while enabling deployment as layer-3 flow overlays atop the current IP network.

The transparency of the Internet has greatly encouraged innovation in the end-hosts and facilitated the deployment of successively more complex network-agnostic applications and services. This accounts for the second area of recent rapid innovations, namely the application layer (e.g. overlays, peer-to-peer, multimedia, content distribution networks, games, VoIP [34], IPTV, etc). The world-wide web has been particularly conducive to the introduction of many new applications (e.g. YouTube, Facebook, Google Maps, etc.) through the adoption of development frameworks such as Web2.0 or AJAX.

In fact, we have seen several trends at the application layer. First, applications are more content-rich than in the past. The importance of audio and video in the global Internet is growing beyond the intra-ISP deployments to support VoIP and IPTV. Content is often sourced from multiple servers that reside in the same data center or even in multiple data centers. These data centers are now a key part of the Internet and source more and more content. Data centers can belong to a single company (for example, Google), but there is a growing trend towards data centers such as Amazon EC2, Microsoft Azure and OVH that dynamically rent processing to external users. Their infrastructure is mainly composed of Ethernet switches and x86 servers. Most network specialists expect such data centers to serve an increasing fraction of the content in the future. Another trend is mashup applications, built by composing several applications that are hosted by different providers. This allows developers to enrich their application with information coming from several providers; applications built on top of Google Maps and Facebook are popular examples. CHANGE will allow operators and application developers to make better use of these data centers by pushing flow-processing capabilities within the data center itself. However, service innovation has often occurred at the application layer not by choice, but as a way to bypass the stumbling block to changes that the instrumental TCP/IP network layer represents. Of course, while introducing new concepts at the application layer simplifies deployment, the solutions are often far from ideal and can exhibit redundancy and sub-optimality resulting from competing objectives.

Cisco's Unified Computing [19] initiative aims to provide an enhanced service-oriented architecture for data centers. By combining switching, server hardware, storage and virtualization in the same device and adding a unified management framework, Cisco's unified computing aims to reduce costs and complexity for data centers, consolidating and integrating components that were traditionally dissociated. Further, it facilitates the provisioning and deployment of new services. Nevertheless, this integration preserves the existing networking model of servers as end-hosts. With CHANGE we go much further and push support for processing not only within the data centers, but also within the network by building on-path flow processing, realized as a composition of processing pipelines, which would otherwise be made up of several specialized boxes. In other words, CHANGE provides a novel way for building complex networks and thus enabling Internet innovation by enabling the integration and convergence of the components functionality.

A network flow-processing platform such as proposed by CHANGE therefore appears to be a good match to meet the varied demands for evolution of the Internet. Indeed, the flexibility of the CHANGE platform is poised to play a crucial role in answering calls for network customization for the future Internet (through the run-time composition of flow-processing modules) to green networking (through the use of virtualization and associated module migration for better power control). There is no doubt that the advent of future network applications will create yet unforeseen communication needs and requirements on the network. We believe the future Internet Architecture must be sufficiently agile to seamlessly accommodate such future requirements. By proposing a paradigm shift towards network processing as a software architecture, CHANGE will provide an agile and flexible base that will allow network providers to quickly meet new communication requirement. In fact, CHANGE will enable the deployment of inherently flexible service delivery infrastructures, significantly simplifying long term network provisioning and planning and thus helping reduce ISPs costs while increasing their competitiveness.

6.2 Survey of Recent Research

Many researchers have recently observed the shortcomings of the current Internet and tried to address them. There is a vast body of research on how to fix the different perceived problems with the current Internet architecture, such as [28, 54, 32, 22, 38, 31, 1, 8, 45, 7, 21, 14, 15, 52] to name just a few. The problems attacked in these works are numerous; we enumerate here the most prominent ones:

- Making the Internet more robust [21] and/or more transparent by including middle-boxes in the Internet architecture [32, 54].

- Supporting seamless mobility, multi-homing, fixed identities whether at the network level [22, 43] or at the content level [31, 38, 7].

- Optimizing access to content [31, 38] by treating content as a network entity.

- Supporting accountability at different levels [1, 14].

- Security - resilience to DDos attacks [1, 8, 45, 14, 58].
- Multicast [45, 9, 36].
- Anycast [31, 38, 45, 29, 51].
- Sharing the internet capacity [14, 52].

Although the problems are diverse, the basic techniques used to solve them seem to revolve around a relatively small toolset:

- Indirection [22, 21, 43, 29, 51].
- On-path and off-path signalling [22, 32, 54, 34].
- Locator/ID split [31], typically with a flat ID space [38, 22, 54], or even without locators [15].
- Name based routing [31, 38, 15].

Despite this huge research effort few of these techniques have been deployed and even fewer are currently used. The problem is twofold; first, for many problems there is currently no clear business case at Internet scale (e.g., multicast, anycast, or name based routing). For the pressing problems, such as security or capacity sharing, point solutions have been deployed and applied as patches in individual operator networks, or indeed at application layer, hence there is no pressing need for a global, optimal solution. The main effect of this myopic approach to problem solving is Internet ossification, as we have also pointed out elsewhere.

Further, for the vast majority of network layer solutions, the biggest barrier has been bootstrapping deployment, as there is no incentive to deploy unless early adopters get a benefit. Early adopters bear most of the costs, so they must be able to capitalise on the benefits; if benefits only come from a critical mass of deployment, early adopters gain nothing. These deployment hurdles exist not only for clean-state approaches, that require changing all (or a significant fraction of) routers [31, 38, 22, 54, 1, 45], but also solutions that require modest changes at the IP layer [14].

Application layer solutions such as I3, OASIS, Overcast, ALM, or RON are relatively easy to deploy, but face other hurdles. By definition these services are network and operator agnostic, and therefore suboptimal in many respects. For instance they do not (and indeed cannot) consider operator policy, or network topology when deciding how to provide a service. Compared to their network layer counterparts application layer solutions end up using suboptimal paths (e.g. i3, RON, OASIS, etc.), and/or transferring more data (e.g. ALM). If used extensively these solutions will effectively fight the network, and the network will respond as it did with peer-to-peer traffic: downgrade service by embedding even more application knowledge in the network.

In contrast, CHANGE has a clear deployment path, but it also inherits most of the desirable properties of network-based solutions. Deployment of a single CHANGE platform should require not much more intervention that deploying a DPI box today. Once such a platform is installed, it can be used within the operators network in many ways, and can even be rented out to customers for custom processing. Hence, deploying such a platform can not only simplify the management and the evolution of the operational network, but also provide additional income (with a business model similar to cloud computing). Once enough individual platforms are deployed, interconnection benefits become obvious, and we expect there will be a strong push for unified standards and platform cooperation. The second key difference is that CHANGE does not preach running every packet through a flow-processing platform: we envision most of the added functionality will only be needed for a comparatively small number of flows, leaving the bulk of the network to do what its best at: moving packets. For instance, DDoS defence requires filtering at a few vantage points in any given DDoS attack, at any given moment in time; this is in contrast with most other proposals [8]. Not only does this ensure the whole platform scales, but it also limits the initial investment needed to get things going. Finally, CHANGE includes as basic primitives both indirection (virtual nets) and on-path processing (signalling). We believe these two mechanisms are fundamental enough to power solutions to most of the aforementioned problems. CHANGE itself does not set out to solve all the above problems; instead, it extends the Internet architecture just enough that proper solutions can be deployed when needed.

6.3 Evolution of the CHANGE Architecture

The CHANGE architecture has been evolving in the past years, benefiting from inputs from internal and external sources.

Internally, the work on MPTCP has spurred the need for invariants, and these have become a central point in the CHANGE architecture, creating an evolvable framework within which end-users can create flow processing whose properties they can understand.

Development work on the platform primitives (WP3) and on the CHANGE inter-platform protocols and client-side API (WP4), together with the concept of invariants, have helped crisply defining flow naming in CHANGE .

Recent publications show the benefits of running middleboxes on commodity hardware [49] and of "outsourcing" middleboxes to the cloud [3]. These works are aligned with the CHANGE vision but focus on a different aspects of the problem space. In particular, two important missing pieces are dealing with security hazards and the problem of understanding flow processing compositions. CHANGE has focused on both of these as they are instrumental in getting the flow processing vision deployed widely.

7 Conclusions

Common knowledge holds that the Internet has ossified because of the many middleboxes that have been deployed worldwide. In this deliverable we have described an accurate picture of this ossification, focusing on the TCP protocol. This work has been presented at the ACM Internet Measurement Conference in 2011 and we find that a third of the paths we probed do keep state at layer 4 or above, and they have the potential to affect future TCP extensions.

Just how difficult is it to extend the Internet today? We have provided an extensive report on our on-going experience in standardizing Multipath TCP for the past four years. The design process has been a lot more complicated than we initially expected and most of that complexity is due to the various middleboxes that affect TCP traffic worldwide. Work will be presented at the Usenix Networked Systems Design and Implementation (NSDI) conference in April 2012.

While MPTCP does work, the MPTCP protocol design is quite complex and defensive, aiming to cover many possible misbehaviours from middleboxes. The main reason for this is the lack of a specified contract with the network; MTPCP tries to find out what the contract is from the endpoints.

Flow processing is the way to break the innovation log jam affecting the Internet today. It provides powerful tools to end-points and operators alike to evolve the network in a transparent and principled way. Security is paramount in flow processing; surely a deployed but insecure flow processing platform will make the Internet worse than it is today.

In this deliverable we have shown how the CHANGE vision can be realised in practice. Our architecture is built using operator-run, flexible flow processing platforms that traffic sources and destinations can use to instantiate processing on their traffic.

Security, correctness and practicality constraints limit the ways flow processing can be performed. We have reasoned about and described a set of rules principles that prescribe what should and should not be allowed in a CHANGE platform. These principles dictate that only the traffic source and destination or parties delegated by them are allowed to instantiate processing. Further, changing IP addresses of traffic in the network is only allowed when the owner of the address agrees.

An important goal of CHANGE is to create a predictable network. To reach this goal we propose the novel concept of *network invariants*. These are dynamic contracts between the users and the network: the users specify what processing should not be applied to their packets, and the network implements these invariants as efficiently as possible.

Despite these restrictions, CHANGE is a flexible platform. We have discussed how we can implement a number of interesting use-case scenarios that are very difficult to instantiate today.

Bibliography

[1] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable internet protocol (aip). In *Proc. ACM SIGCOMM*, Seattle, WA, USA, August 2008.

[2] K. Argyraki, A. Baset, B-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, and S. Nedveschi. Can software routers scale? In *Proc. PRESTO*, Seattle, WA, USA, August 2008.

[3] Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. Justine sherry and shaddi hasan and colin scott and arvind krishnamurthy and sylvia ratnasamy and vyas sekar. In *Proc. SIGCOMM*, 2012.

[4] T. Aura. *Cryptographically Generated Addresses (CGA), RFC 3972*. IETF, mar 2005.

[5] A. Bakre and B. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. In *Proc. IEEE ICDCS*, pages 136–143, 1995.

[6] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz. Improving TCP/IP Performance over Wireless Networks. In *Proc. ACM MOBICOM*, pages 2–11, 1995.

[7] Hari Balakrishnan, Karthik Lakshminarayanan, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Michael Walfish. A layered naming architecture for the internet. In *Proc. ACM SIGCOMM*, Portland, OR, USA, August 2004.

[8] Hitesh Ballani, Yatin Chawathe, Sylvia Ratnasamy, Timothy Roscoe, and Scott Shenker. Off by default! In *Proc. ACM Workshop on Hot Topics in Networks (Hotnets)*, Maryland, MD, USA, November 2005.

[9] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proc. ACM SIGCOMM*, Pittsburgh, PA, USA, August 2002.

[10] L. Berger. *Generalized Multi-Protocol Label Switching (GMPLS) Signalling Functional Description, RFC 3471*. IETF, January 2003.

[11] Andrea Bittau, Michael Hamburg, Mark Handley, David Mazières, and Dan Boneh. The case for ubiquitous transport-level encryption. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 26–26, Berkeley, CA, USA, 2010. USENIX Association.

[12] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *An Architecture for Differentiated Services, RFC 2475*. IETF, dec 1998.

[13] R. Braden, D. Clark, and S. Shenker. *Integrated Services in the Internet Architecture: an Overview, RFC 1633*. IETF, jun 1994.

[14] Bob Briscoe and Alessandro Salvatori. Policing congestion response in an internetwork using re-feedback. In *Proc. ACM SIGCOMM*, Philadelphia, PA, USA, August 2005.

[15] Matthew Caesar, Tyson Condie, Jayanthkumar Kannan, Karthik Lakshminarayanan, Ion Stoica, and Scott Shenker. Rofl: routing on flat labels. In *Proc. ACM SIGCOMM*, Pisa, Italy, September 2006.

[16] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. *RFC 3234*, Feb. 2002.

[17] R. Chakravorty, S. Katti, J. Crowcroft, and I. Pratt. Flow Aggregation for Enhanced TCP over Wide-Area Wireless. In *Proc. IEEE INFOCOM*, pages 1754–1764, 2003.

[18] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *Proceedings on Advances in cryptology*, CRYPTO '88, pages 319–327, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[19] Cisco. Cisco, unifed computing, http://www.cisco.com/en/US/netsol/ns944/index.html.

[20] David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden. Tussle in cyberspace: defining tomorrow's internet. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '02, pages 347–356, New York, NY, USA, 2002. ACM.

[21] M. Frans Kaashoek Robert Morris David G. Andersen, Hari Balakrishnan. Resilient overlay networks. In *Proc. ACM Symposium on Operating Systems Principles SOSP*, Banff, AL, Canada, October 2001.

[22] L. Eggert and M. Liebsch. *Host Identity Protocol (HIP) Rendezvous Mechanisms, draft-eggert-hip-rendezvous*. IETF, July 2004.

[23] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high-performance virtual routers on commodity hardware. In *Proc. CoNEXT*, Madrid, Spain, December 2008.

[24] D. Estrin, T. Li, Y. Rekhter, K. Varadhan, and D. Zappala. *Source Demand Routing: Packet Format and Forwarding Specification (Version 1), RFC 1940*. IETF, may 1996.

[25] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. IP options are not an option. *Tech. Rep. UCB/EECS- 2005-24*, 2005.

[26] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural guidelines for multipath TCP development. *RFC 6182*, Mar. 2011.

[27] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP extensions for multipath operation with multiple addresses, Jul 2012. IETF draft (work in progress).

[28] P. Francis and R. Gummadi. Ipnl: A nat-extended internet architecture. In *Proc. ACM SIGCOMM*, San Diego, CA, USA, August 2001.

[29] Michael J. Freedman, Karthik Lakshminarayanan, and David Mazires. Oasis: anycast for any service. In *USENIX OSDI*, San Jose, CA, USA, May 2006.

[30] Adam Greenhalgh, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. Flow processing and the rise of commodity network hardware. *SIGCOMM Comput. Commun. Rev.*, 39:20–26, March 2009.

[31] M. Gritter and D. Cheriton. An architecture for content routing support in the internet. In *USENIX OSDI*, San Francisco, CA, USA, March 2001.

[32] S. Guha and P. Francis. An end-middle-end approach to connection establishment. In *Proc. ACM SIGCOMM*, Kyoto, Japan, August 2007.

[33] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, 2001.

[34] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. *SIP: Session Initiation Protocol, RFC 2543*. IETF, March 1999.

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

[36] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole. Overcast: reliable multicasting with on overlay network. In *USENIX OSDI*, San Diego, CA, USA, October 2000.

[37] J.Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. *RFC 3135*, Jun. 2001.

[38] T. Koponen, M. Chawla, B. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *Proc. ACM SIGCOMM*, Kyoto, Japan, August 2007.

[39] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[40] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. *RFC 2018*, Oct. 1996.

[41] OpenFlow. The openflow switching, http://www.openflowswitch.org/.

[42] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. *RFC 2663*, Aug. 1999.

[43] C. Perkins. *IP Mobility Support for IPv4, RFC 3344*. IETF, August 2002.

[44] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the Narrow Waist of the Future Internet. In *Proc. ACM Hotnets*, 2010.

[45] L. Popa, I. Stoica, and S. Ratnasamy. Rule-based forwarding (rbf): Improving internet's flexibility and security. In *Proc. ACM Workshop on Hot Topics in Networks (Hotnets)*, New York, NY, USA, November 2009.

[46] E. Rosen, A. Viswanathan, and R. Callon. *Multiprotocol Label Switching Architecture, RFC 3031*. IETF, January 2001.

[47] S. Savage. Sting: a TCP-based Network Measurement Tool. In *USENIX USITS*, 1999.

[48] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *ACM CCR*, 29(5):71–78, 1999.

[49] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. NSDI*, 2012.

[50] R. Stewart, M. Ramalho, and et al. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. *RFC 3758*, May. 2004.

[51] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. *IEEE/ACM Transactions on Networking*, 12(2):205–218, April 2004.

[52] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Transactions on Networking*, 11(1):33–46, February 2003.

[53] Jon Turner, Patrick Crowley, John Dehart, Amy Freestone, and Fred Kuhns. Supercharging planetlab a high performance, multi-application, overlay network platform. In *Proc. ACM SIGCOMM*, Kyoto, Japan, August 2007.

[54] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *USENIX OSDI*, San Francisco, CA, USA, December 2004.

[55] D. Watson, M. Smart, G. R. Malan, and F. Jahanian. Protocol Scrubbing: Network Security Through Transparent Flow Modification. *IEEE/ACM ToN*, 12(2):261–273, 2004.

[56] D. Wetherall, U. Legedza, and J. Guttag. Introducing new internet services: Why and how. *IEEE Network Magazine*, 12(1):12–19, August 1998.

[57] D. Wetherall, U. Legedza, and J. Guttag. K. clavert and s. bhattacharjee and e. zegura and j. sterbenz. *IEEE Communications Magazine*, 36(10):72–78, October 1998.

[58] A. Yaar, A. Perrig, and D. Song. Siff: A stateless internet flow filter to mitigate ddos flooding attacks. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2004.

[59] Xiaowei Yang, David Wetherall, and Thomas Anderson. A dos-limiting network architecture. In *Proc. ACM SIGCOMM*, 2005.