Architectural Support for Managing Privacy Tradeoffs in the Internet

David Naylor

August 2017
CMU-CS-17-116

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Peter Steenkiste, Chair
Vyas Sekar
Srini Seshan
Dave Oran *(Network Systems Research & Design; MIT Media Lab)*
Adrian Perrig *(ETH Zürich)*

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program, by the National Science Foundation under grants numbered CNS-1040801 and CNS-1345305, and by the European Union under the FP7 Grant Agreement n. 318627 (Integrated Project "mPlane").

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

Keywords: networks, privacy, anonymity, secrecy, accountability, TLS, HTTPS, encryption, middleboxes, trusted computing, SGX

To my parents
who made me think getting a PhD
was a normal thing to do.

Abstract

Using a communication network entails an inherent privacy risk: packets cross an infrastructure maintained by several parties other than the sender and receiver, each of which has the opportunity to observe the packets as they are processed and forwarded. This poses a risk because packets carry information that users might rather keep private, namely: (1) the source address, which exposes the sender, (2) the destination address, which exposes the recipient, and (3) the body, which can expose user data. Beyond the information explicitly carried by the packet, observers can also learn sensitive things merely from the fact that a packet happened to be in a certain place at a certain time. All of this information is often divided into two categories: **data** (the actual message being communicated, e.g., the contents of an email) and **metadata** (information *about* the communication, e.g., "*A* emailed *B* at 12:07 today").

Fortunately, we have tools, widely used in practice, to protect this information. Unfortunately, these tools tend to make aggressive trade-offs, sacrificing other desirable properties for the sake of privacy. For example, to protect data, the use of encryption is widespread—on the Web, for instance, many sites have switched from HTTP to HTTPS. Unfortunately, encryption blinds **middleboxes**, which can lead to a loss of *functionality*, *performance*, and even *security*. And to protect metadata, anonymous communication systems like Tor reduce *accountability* by preventing network operators from learning who sent a packet and also often introduce *performance* overheads.

These "privacy vs. X" tussles seem fundamental, because privacy requires *hiding* information like source addresses and payloads, while the other properties—performance, accountability, functionality, and security—require *exposing* that information. How can we do both? In this thesis, we argue first that a practical balance *is* possible if we *carefully control access to packet data and metadata* and second that this requires *architectural support from the network*.

We make this argument in two parts. First, we show how to keep in-flight data private while at the same time allowing middleboxes like caches, compression proxies, and intrusion detection systems to operate. We motivate, design, and evaluate two protocols for secure communication that includes middleboxes, each one granting data access only to middleboxes explicitly trusted by an endpoint and also limiting the scope of what those middleboxes can do with that data. With fully-functional implementations, we show that these protocols are deployable and have minimal performance overhead.

Second, we show how re-thinking the way the network treats source addresses can enable a balance between privacy and accountability that is not possible today. We present the design of a new network architecture that separates source addresses into distinct "accountability" and "return" addresses and show with trace-driven analysis that the performance overhead is reasonable. In order to compare our new architecture to related work, we also develop an evaluation methodology for quantifying "how private" a network architecture is.

Acknowledgments

This thesis would undoubtedly not have happened without the support, guidance, patience, and friendship of a long list of people. I will unsuccessfully attempt to enumerate them here; to anyone I've inadvertently left out, my apologies.

To start, huge thanks to my advisor, Peter Steenkiste, for six years of guidance and mentorship (though even after six years, I still haven't managed to pick up his ability to immediately see through superfluous details and articulate the heart of a problem, no matter how badly I explain something). I particularly appreciated his willingness to leave me to my own devices when it came time to format papers or design slides. The rest of my committee deserves thanks as well: Vyas Sekar, for being a top notch random idea generator and evil question simulator; Srini Seshan, for his honest appraisal of my color schemes; Dave Oran, for not giving up on me after I completely misunderstood his question after my talk at SIGCOMM; and Adrian Perrig, for consistently reminding us networking folks how security works.

I'd also like to thank my collaborators outside CMU, starting with the crew at Telefónica. Thanks Dina Papagiannaki, Matteo Varvello, Jeremy Blackburn, Ilias Leontiadis, Yan Grunenberger, Alessandro Finamore, and Pablo Rodriguez for an amazing summer (and winter!) in Barcelona (and also for *not* naming mcTLS "TruMP"—we really dodged a bullet there). And to my MSR colleagues Thomas Karagiannis and Christos Gkantsidis for an equally awesome, albeit less sunny, summer in Cambridge.

Even with the support of all these amazing researchers, no actual reserach would happen without the additional support of SCS, CSD, and XIA staff. Deb Cavlovich, Jenn Landefeld, Angie Miller, Kathy McNiff, Angy Malloy, Dan Barrett, and Nitin Gupta—sometimes I think yinz are the only ones in the building who actually know what's going on.

Beyond getting *through* the PhD, I could not have gotten *to* it without the help of my undergraduate research mentors—Alberto Segre, Ted Herman, Phil Polgreen, Sriram Pemmaraju, Geb Thomas, and the rest of the CompEpi team; my family, for raising baby-David in an environment that valued education, curiosity, and rational thought; and the privilege that comes with being a white, male American.

Finally, shoutouts to my Pittsburgh friends, who deserve credit for keeping me happy and sane for the past six years. Alphabetically, thanks to: Alex Beutel, for giving us all a hard time when we tried to work on Friday nights; Brock, for being the cutest dog ever, even when he bites you because you have a deadline so you can't play with him; JP Dickerson, for giving us one more reason to not use Comic Sans; Zakir Durumeric, for scanning the whole Internet for me; Lili Ehrlich, for all the awesome tattoo suggestions; Nico Feltman, for backing me up when I'm being pedantic and (I guess) for coining "Nay-Nay"; Sam Gottlieb, for distracting me from research with Teddy; Dongsu Han, for being a vim super-user role model; Sophie Hood, for teaching me about all the weird colors vegetables come in; Sid Jain, for briefly keeping me company in GHC 7509; Angela Jiang, for making me feel less guilty about my Uber usage; Hyeontaek Lim, for explaining to me how systems actually work; Sarah Loos, for the grad lounge; Dana and Yair Movshovitz-Attias, for never complaining that we

always addressed them as a unit; Matt Mukerjee, for at least somewhat successfully convincing me not to spend all my time working and also for keeping me from being a hoarder; Kenton Murray, for making sure at least one of us has cool hair; George Nychis, for helping me decipher Peter's handwriting; Vagelis Papalexakis, for single-handedly raising CSD's average height by like at least an inch; Alex Poms, for reminding me to be thankful I'm acclimated to winter; Nic Resch, for teaching us Canadian; Wolf Richter, for showing us that reserach code doesn't have to be research code; Raja Sambasivan, for encouraging me to ignore ugly conference formatting guidelines; Nick Sharp, for pairing applesauce with weird things; Evan Shimizu, for literally being the SCS dragon; Richard Wang, for only occasionally stalking me with Cobot; Gabe Weisz, for keeping the department stocked with quality beer; Colin White, for encouraging healthy activities like eating donuts in the middle of a run; David Witmer, for running two awesome visit days with me and for making me look stronger by association at the gym; John Wright, for being right about expander graph networks all along; Yuchen Wu, for forcing me to switch to zsh; Yu Zhao, for not leaving me by myself in the office; Jayant, Kevin, Shiva, and Patrick, for running the CS party house; and to Duffy, Molly, Greg, Kevin, Randy, Kristy, Arush, Pallavi, Su Su, Mariah, Hank, Dana, Alex, Elizabeth, and Matteus, for reminding me there is life outside CS.

Contents

List of Figures

List of Tables

Part I

Introduction and Background

Chapter 1 Introduction

Before the Internet, we stored all of our personal (digital) data on our personal computers, which we kept safely in our homes or offices. That data was only accessible to others in person—they could read our files only if we typed in our passwords and permitted them physical access to our machines. Understanding who could see our data was easy; sharing it was not.

The Internet changed this—it made sharing trivial, but it brought with it many new privacy concerns for users. Using a communication network inherently involves sending data across an infrastructure maintained by a number of parties other than the sender and receiver, each of which has the opportunity to observe packets as they are processed and forwarded. This poses a risk because packets carry information that users might rather keep private, namely: (1) the packet's source address, which exposes the *sender*, (2) the destination address, which exposes the *recipient*, and (3) the body, which can expose various pieces of user data. And this is just the information explicitly carried in the packet; observers can also learn sensitive information merely from the fact that a packet happened to be in a certain place at a certain time (exploiting this information is known as **traffic analysis**). All of this information is often divided into two categories: **data** (the actual message being communicated, e.g., the contents of an email) and **metadata** (information *about* the communication, e.g., "*A* emailed *B* at 12:07 today").

There are good reasons to hide both types of information. For example, preventing unauthorized access to data sent over the network protects online shoppers' credit card information from theft and allows employees of a company to exchange confidential documents. Other times, *what* users send is less important than *to whom* they send it, so techniques for hiding metadata help users protect their identities while, for instance, accessing health websites without revealing personal medical conditions, posting to whistleblowing websites, or speaking out against an oppressive political regime [214].

Well-known (and widely used) techniques exist today that provide these properties (we describe them in more detail later). The problem with these existing methods—and the focus of this thesis—is that they tend to make aggressive trade-offs, sacrificing other desirable properties for the sake of privacy. For example, to protect data, the use of encryption is widespread—on the Web, for instance, many sites have switched from HTTP to HTTPS. Unfortunately, encryption blinds **middleboxes**, which, as we will discuss, can lead to a loss of *functionality*, *performance*, and/or *security*. To protect metadata, anonymous communication systems like Tor reduce *accountability* by preventing network operators from learning who sent a packet and often introduce *performance* overheads.

These *Privacy vs. X* tussles are fundamental, so striking a balance is non-trivial. In some cases, users themselves might have multiple conflicting objectives. For instance, users want privacy *and* performance, so they might be torn about whether or not to give a compression proxy access to

their data. Worse, the Internet is comprised of many different entities, each with its own goals and priorities. For example, *users* may want to use the network anonymously, while *operators, employers, and law enforcement* want to keep their networks secure and prevent misbehavior. Likewise, allowing middleboxes access to packet payloads is tricky because *users* want to keep their information secret, *operators* want security and low cost, and *content providers* want to deliver a good user experience.

This all adds up to a formidable challenge: designing privacy mechanisms that offer a *practical* balance between privacy, functionality, accountability, performance, and security. Viable solutions must provide enough of each property to satisfy all parties.

THESIS STATEMENT

Despite the fundamental tension between hiding personal information from the network (to provide privacy) and exposing it (to enable accountability, performance optimizations, additional functionality, and security screening), carefully designed architectural support can enable mechanisms that offer realistic balances.

1.1 Background

1.1.1 Our Setting

This thesis considers privacy and security issues related to data transmitted over a communication network. Before we discuss those privacy and security properties themselves, this section lays out some basic assumptions about the network and the adversary. Individual chapters may make (and will state if so) different or additional assumptions.

We model a network as a set of **hosts**, or **endpoints**, and **network boxes** (e.g., routers, NATs, firewalls, etc.) connected by **links**. A "conversation" between two endpoints is a **flow**, which consists of one or more **packets**.

The adversary might have **vantage points** on any number of links or boxes—for example, by sniffing a Wi-Fi link, by compromising a router, or by subpoenaing logs from one or more routers. Every time a packet passes through a vantage point, an event we call a **sighting**, the adversary can observe and/or modify any part of the packet. The adversary can also drop packets entirely, reorder packets, or inject brand new packets of its own.

We assume that the adversary cannot break standard cryptographic primitives (i.e., it is computationally bounded) and that it cannot compromise the endpoints themselves (that is, we do not consider privacy from endpoint applications, like web servers, who might track a user's activity).

1.1.2 User Goals

What security and privacy properties do users care about when they use the Internet? To start, let us consider the problem from an intuitive perspective. Broadly speaking, users have three goals (shown on the right half of Figure 1.1):

> **USER GOAL 1: Protect their personal information from disclosure.**
> For example, users want to be able to read their email or deposit a check without anyone else (aside from their email provider or their bank) learning the contents of their inbox or their checking account.

> **USER GOAL 2: Make sure the information they access is the "correct" information.**
> For example, drivers navigating with Google Maps should be sure that the route they follow truly came from Google and investors trading stocks cannot afford to make decisions based on bogus market data.

> **USER GOAL 3: Prevent others from tracking their online activity.**
> For example, an employee may not want their employer or health insurance provider to know what symptoms they have researched recently and a reporter might want to communicate with sources while keeping them anonymous.

Now let us be more precise. First, we can draw a line between the first two user goals and the third: the former relate to the *data* being communicated, and the latter relates to the *communication* itself. Next, we will introduce the five standard security properties that underlie the user goals: **secrecy**, **entity authentication**, **data authentication**, **sender/receiver anonymity**, and **flow unlinkability**. The left half of Figure 1.1 shows these properties; an arrow from a property to a user goal indicates that that property is required to meet that goal. Throughout the rest of this thesis, we will extend this figure, working from right to left. In Part II, we describe the first three properties (those needed for data privacy) in more detail and introduce techniques for providing them. Likewise, Part III explains the latter two properties (those needed for metadata privacy) and discusses techniques that offer them. Each part will focus on how these techniques can hinder performance, accountability, functionality, and security for the sake of privacy and how appropriate network support can resolve this tension. Figure 1.2 shows the final, completed version of Figure 1.1.

1.2 Overview and Contributions

1.2.1 Protecting In-Flight Data (Without Giving Up Performance and Functionality)

To protect data sent over the network, communicating parties can use **encryption**. This is often done at the transport layer with *Transport Layer Security (TLS)* [94], the standard protocol for

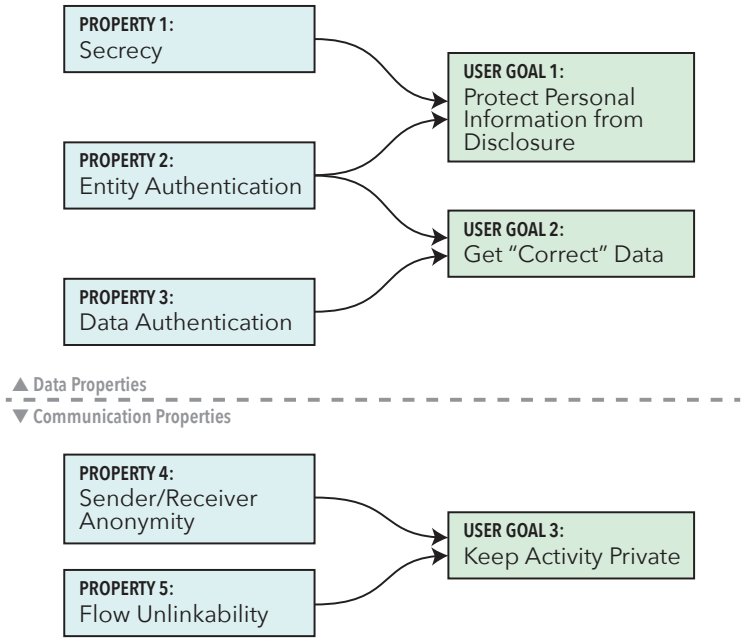


Figure 1.1: **Properties and Goals.** Five key security and privacy properties underlie users' privacy goals.

adding encryption and authentication to a TCP connection. Most notably, the secure version of HTTP, known as HTTPS, is simply HTTP on top of TLS instead of directly on top of TCP. Encryption can also be performed at the network layer using IPsec [147].

Encryption guarantees that only the communicating parties can see the contents of their communication, but this is meaningless without a way to verify that the encrypted communication is with the intended party. That is, when a user makes a purchase on Amazon.com, how do they know the server to which they just sent their credit card number is actually an Amazon server? To solve this problem, encryption is often used in conjunction with an **authentication** mechanism. In the Web, each site has a public/private key pair, and a **public key infrastructure (PKI)** consisting of a hierarchy of **certificate authorities (CAs)** maintains a public mapping of public keys to domain names. These mappings are represented by **certificates**, which are digitally signed statements issued by CAs that a particular domain owns a particular public key. During a TLS handshake, the client's Web browser verifies that the server has access to the private key belonging to the intended domain. Assuming sites truly keep their private keys private, this is evidence that the client has established a connection with the intended party.

Limitations. Encryption can reduce performance, functionality, and, ironically, security by preventing **middleboxes**—devices in the network that process traffic beyond basic packet forwarding—from operating on packet contents. Middleboxes are widely deployed (most networks have roughly the same number of them as they do routers and switches [208]), and they perform a variety of useful functions. For example, caches can reduce latency by storing a copy of recently requested content so it can be returned faster for subsequent requests; compression proxies help users manage mobile data usage by compressing data; firewalls, intrusion detection systems (IDSes), and virus scanners enhance security; load balancers improve performance by spreading traffic among multi-

ple backend servers; and parental filters help parents control what young children can access. When traffic is encrypted, these middleboxes become blind—all they can see is a stream of encrypted bits whizzing past—and we lose the benefits of the services they provide.

This problem has attracted a fair amount of attention lately. There are a number of proposed techniques (some already used in practice) for including middleboxes in encrypted sessions [99, 156, 159, 164, 187] (see Section II.3 for more details) as well as ongoing discussions in groups like the IETF, IEEE, and ETSI. Unfortunately, these solutions are ad hoc and lead to several important security problems.

As an example, let us consider the most common approach for adding a middlebox to a TLS connection. Suppose a company wants to virus scan all PDFs downloaded on the company network. Before a new employee is given network access, an administrator installs a custom root CA certificate on that employee's machine. Now, every time the employee opens a new TLS connection from that machine, the company's virus scanner intercepts the connection, fabricates a certificate for the intended Web site on-the-fly (which the browser will accept because it is signed by the custom root certificate), and opens a second connection to the target server. The browser (and the employee) thinks it has an end-to-end secure connection to the server, when in reality, the virus scanner can inspect and modify all traffic completely transparently to the endpoints. With this type of interception, is it impossible for the client to (1) authenticate both the middlebox and the server (it must simply trust that the middlebox will properly authenticate the server) and (2) ensure that the middlebox↔server connection is using a particular cipher suite (or that it is using encryption at all). Indeed, both of these problems occur in practice with alarming regularity: interception proxies use outdated cipher suites or fail to check server certificates [99].

Approach. The problems above stem from the fact that techniques like interception-with-custom-root-certificate all try to retrofit TLS, which was designed to be used between two parties, onto *client-middlebox-server* connections. Since middleboxes have become an integral part of modern network infrastructures, we argue that we need a new protocol that acknowledges this reality and supports "multi-entity communication" by design. Part II describes our efforts toward designing such a protocol, which are guided by two core principles:

Middleboxes as First-Class Citizens. TLS was designed to facilitate a secure connection between two endpoints and thus is not equipped to allow an endpoint to authenticate or negotiate ciphers with more than one other party. This leads to the problems described above. Instead, we need a protocol that allows endpoints to explicitly perform a TLS-like handshake with both the other endpoint as well as one or more middleboxes.

Principle of Least Privilege. In some cases, endpoints may wish to restrict how much of the session data a middlebox can see or what types of modifications it can make. For example, an administrator might require a user to give access to a middlebox the user does not completely trust. Or, even if the user does fully trust the middlebox, its presence increases the attack surface (it is yet another party that could be attacked by an adversary trying to steal user data). Since most middleboxes only need access to specific portions of the session data, or only need to perform specific operations on that data, it is possible to lessen the impact of a potential breach by limiting what the middlebox can see and do. (In general, this idea is called the *principle of least privilege*: in any system, each component

should have the fewest permissions it needs to do its job and no more [200].) In Chapter 3 we discuss cryptographic approaches and in Chapter 4 we leverage trusted hardware.

Finally, the broader point we want to make here is that it remains unclear which security properties are required for "secure" communication among many parties. This is an important question for researchers and practitioners to address, and we hope the work presented here represents a useful starting point.

Contributions. We make the following contributions, described in Part II:

- **A study of the impacts of encrypting Web traffic.** In Chapter 2 we present our measurement study on the impact of using TLS, which shows that the loss of middleboxes is the most serious negative effect of switching to HTTPS.
- **A design space for secure multi-entity communication protocols.** Chapter 4 begins with a description of the properties one might want in a secure multi-entity communication protocol and the interactions among them.
- **Techniques for giving middleboxes access to encrypted data with fine-grained access control.** Chapter 3 presents a novel abstraction, *encryption contexts*, which allows endpoints to restrict which parts of the data stream middleboxes can read or write. We then present Multi-Context TLS (mcTLS), a protocol that uses encryption contexts to apply the *principle of least privilege* to middleboxes.
- **Techniques for securely outsourcing middleboxes.** Chapter 4 presents Middlebox TLS (mbTLS), a protocol for secure communication including middleboxes that uses trusted computing hardware, like Intel SGX, to protect session data from middlebox infrastructure providers. mbTLS also solves practical deployment problems, like legacy support and in-band middlebox discovery.

1.2.2 Protecting Communication Metadata (Without Giving up Accountability)

Several anonymity systems exist in research and in practice. The simplest is an **anonymizing proxy**, which works by encrypting the sender's packets and tunneling them to a proxy that decrypts them and forwards them to the ultimate destination. A single proxy is sufficient against an adversary with limited capabilities; if an adversary is more powerful (e.g., has multiple vantage points throughout the network), this idea can be extended by encrypting the packet multiple times and forwarding it through a series of proxies, each of which strips off a layer of encryption and forwards the packet to the next hop. This process is known as **onion routing** [118] and is used by Tor [96].

However, since Tor attempts to introduce as little delay as possible, it is still susceptible to timing attacks—an adversary can link packets entering and leaving a Tor relay even though the payload has been re-encrypted. One solution is for the anonymizing proxies to gather a batch of packets from multiple clients and release them in a different order after an artificial delay. Such a proxy is called a **mix** [75]; a network of mixes can hide true traffic patterns by introducing artificial delay or dummy cover traffic [54, 89, 90, 112, 122, 153, 154, 169, 210].

Another solution is to form an overlay network and run a special cryptographic communication protocol on top where users of the overlay can communicate anonymously (examples of this include **DC-nets** [76, 119] and **mailbox systems** [41, 85, 150, 223] based on private information retrieval). At the IP layer, the adversary only learns that a user is using the system.

(See Section III.1.2 and Section III.3.1 for a more detailed discussion of these techniques and systems.)

Limitations. These systems all share two major limitations. The first is that many impair performance by introducing latency or reducing available bandwidth. These systems offer various degrees of protection, with a pronounced tradeoff between anonymity and performance. This means that many of these techniques are not suitable for default, always-on protection and must be used selectively, only when the need for privacy outweighs the desire for a fast browsing experience.

The second limitation is that, though the particulars vary from system to system, each system is designed to hide a packet's true sender and true receiver. From a privacy perspective, this is exactly the property we want: patients can access healthcare information unobserved, dissidents can safely dissent, and whistleblowers can whistleblow without fear of retaliation. Unfortunately, this also means that those who want to misbehave—e.g., launch network attacks, download illegal content, or harass others—can also do so with impunity. Network administrators and law enforcement, who typically use packet source addresses to find the sources of such abuses, are left more or less helpless. The result is that accountability becomes significantly more difficult, if not impossible.

In fact, the problem is even worse, because, even in the absence of these anonymity systems, today's Internet does not provide strong accountability to begin with. IP source addresses are easily spoofed—that is, one host can easily claim to be a different host. When a host sends a packet, it is free to write any address it likes into the source address field. There are a couple (relatively weak) methods used to deal with this today. First, source networks can perform **egress filtering**—that is, the network's border routers can drop any outbound packets with source addresses outside their address space. Similarly, transit networks can perform **ingress filtering** [48], dropping inbound packets with invalid source addresses. One common way to determine if an incoming source address is valid is called **unicast reverse path forwarding (uRPF)**. Upon receiving a packet on a particular interface, a router checks its forwarding tables to make sure that its route to that source address leaves through that interface. If not, the packet is dropped. More sophisticated variants have been proposed by the research community [98, 141, 185], but have not been adopted in practice.

Unfortunately, ingress and egress filtering are at best marginally effective because they only operate at the granularity of a subnet (hosts within a subnet can still spoof one another's addresses). Furthermore, even if we could completely prevent spoofing, the accountability picture is not complete. Many attacks come from compromised hosts ("bots") that send attack traffic from their actual addresses. Since this traffic is not spoofed, the filtering techniques described above do not help, and IP does not provide any mechanisms for reporting or blocking malicious traffic.

Approach. In this thesis, we argue that both classes of solutions (anonymity systems and anti-spoofing filters) are constrained by *lack of support from the network itself*. In the case of anonymity, in order to hide the identity of a packet's sender, clearly *something* must change about the source

address. Because the IP architecture does not offer any primitives for anonymous communication, the techniques described above all operate as overlays, implicitly changing the meaning of the source address. The IP specification assumes that the source address identifies a sender end-to-end, but this is not true, for example, of a packet leaving a Tor node. Fundamentally, this is the reason anonymity systems subvert accountability. Network operators who only have a network-level view have no way to discover the true sender. Any change to the semantics of source addresses requires support from the network if some degree of accountability is to be maintained.

The same holds true for accountability: without support from the network (i.e., routers), endhosts have limited options for (1) preventing source address spoofing and (2) blocking malicious traffic. (For instance, in the case of DDoS, once the network delivers attack traffic, the damage is done, even if the victim drops it.) We are not the first to point this out—the Accountable Internet Protocol (AIP) [40] proposed the idea of **self-certifying addresses**, where each host has a public/private key pair and its network address is derived from the public key. AIP builds on this with two things: (1) an *anti-spoofing* mechanism (routers can challenge senders to prove they have the private key corresponding to their address) and (2) a *shutoff* mechanism where a victim under attack can tell the attacker's network interface card (NIC) to block the offending flow. AIP's primary drawback is its lack of privacy: hosts are now cryptographically linked to each packet they send, making anonymous communication even harder than it is today.

We propose that architectural support—that is, anonymity and accountability mechanisms directly supported by the network—can more naturally enable realistic trade-offs. So, we ask the question: if we could make changes to the network architecture, how could we balance the two? The answer, at a high level, is to *decouple anonymity and accountability*. Currently, the only "handle" the network provides for finding and punishing a malicious sender is the source address. Unfortunately, the source address is also used to identify the sender (so the receiver knows where to respond). This makes it difficult to have both anonymity and accountability, since for the latter source addresses would ideally be unforgeable, while for the former there would ideally be no source addresses at all. In Chapter 5 we describe the Accountable and Private Internet Protocol (APIP) [176], our network architecture that decouples the source address' accountability and return address roles. To our knowledge, APIP was the first network architecture to provide both more accountability and more privacy than IP. Since then, the Accountable and Private Network Architecture (APNA) [155] builds on these ideas with a thorough design for ISP-brokered accountability and privacy.

In the course of doing this work, we sought to put APIP in context with other anonymity systems and encountered an interesting problem: how can we measure "how private" a network *architecture* is? Existing anonymity metrics, as we describe in Section III.3.2, are geared toward evaluating a complete system—that is, how much anonymity can a user expect given a particular topology and traffic patterns? But how much of this is fundamental to the architecture and how much is the product of the details of one particular deployment of that architecture is hard to say. (No matter how good an architecture is, if there are only two active users, observers know exactly who is communicating.) In Chapter 6 we present our work on a deployment- and usage-independent metric for architectural anonymity.

Contributions. Concretely, we make the following contributions, described in Part III:

- **Architectural support for balancing privacy and accountability.** In Chapter 5 we intro-

duce a new network architecture, the Accountable and Private Internet Protocol (APIP), that explicitly balances privacy and accountability—and, in fact, offers more of each than the current Internet, without introducing significant overhead in the default case.

- **Techniques for quantifying privacy.** While—and after—we designed APIP, we found we struggled to rigorously compare it to other architectures in terms of privacy. This turned out to be a tricky, open problem; Chapter 6 describes our efforts on how to quantify "how much privacy" a network architecture provides.

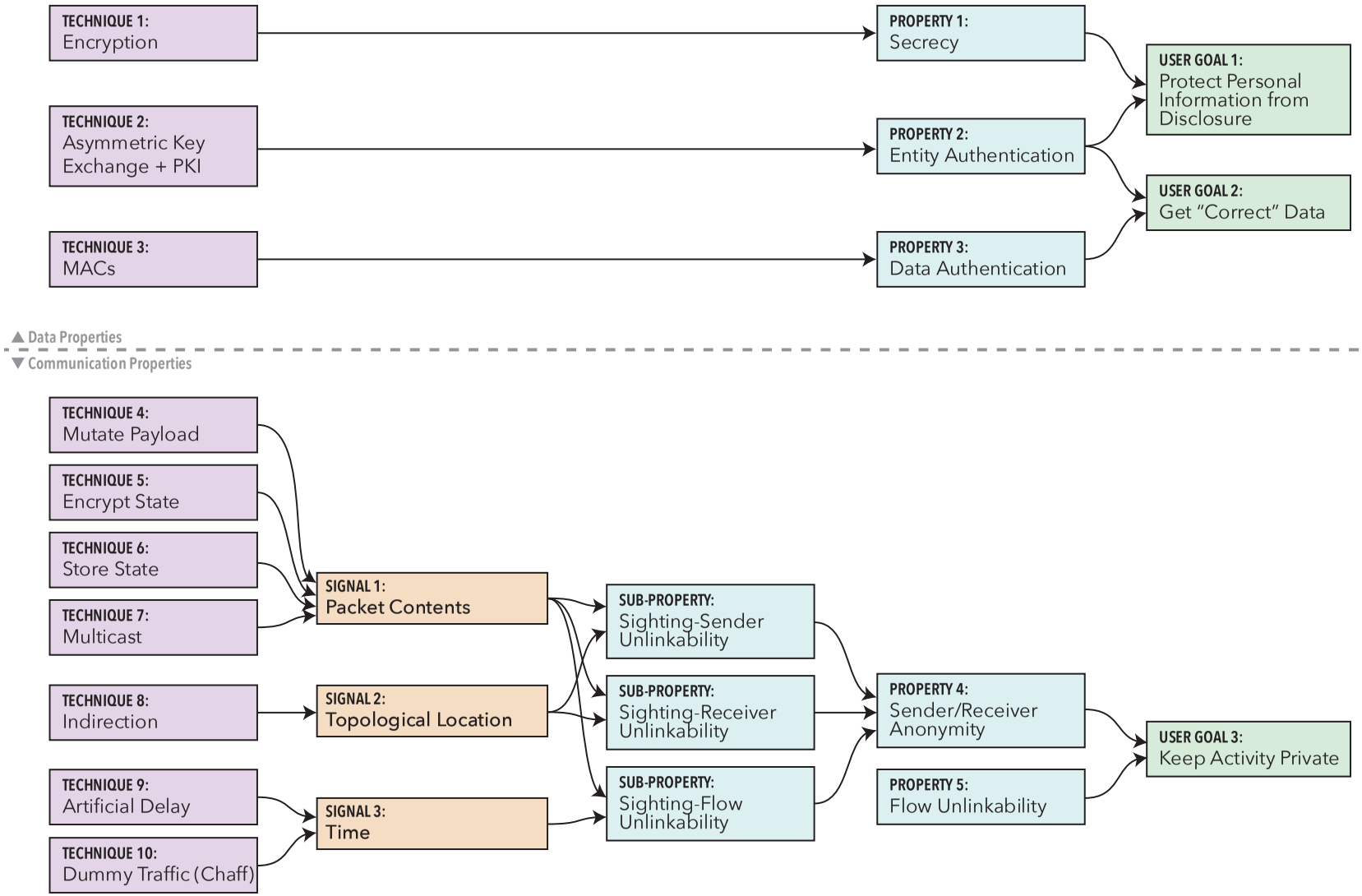


Figure 1.2: **The Big Picture.** A bird's-eye view of the high-level user goals, the precise security properties that underpin them, and common techniques for achieving those properties. This figure is meant as a reference, and will be filled in and explained piece by piece throughout this document.

Part II

Protecting In-Flight Data
(Without Giving Up Performance and Functionality)

1. What are the consequences of encrypting all Internet traffic?
2. How can we keep the benefits of encryption without giving up the benefits of middleboxes?
3. How can we do this even when middleboxes are physically controlled by an untrusted party?

Based on work appearing in CoNEXT 2014 *[175] and* SIGCOMM 2015 *[177].*

Introduction

When users communicate on the Internet, they might naturally expect the data they send to be safe from prying eyes and the data they receive to be "correct" (the first two goals we introduced in Section 1.1.2). Unfortunately, the Internet was not designed to provide these guarantees. It was not designed to protect the data it carries, nor was it designed to make any guarantees about the identities of remote endpoints—it only delivers data to addresses that have no higher-level meaning, and even those are easily faked. Intuitively, this means that an attacker in the network is able to:

1. Read data a user sends or receives.
2. Pretend to be the person with whom the user wants to communicate.
3. Change data a users sends or receives.

As a concrete example, suppose a user makes a purchase from an online vendor. The first vulnerability means that an attacker can see the user's credit card number when the user sends it to the vendor. Even if we fix this by using encryption, the second vulnerability means the attacker can still learn card numbers by pretending to be the vendor. Finally, the third vulnerability means that the attacker can modify the user's shipping address to one of the attacker's choosing.

II.1 Protecting In-Flight Data...

Now we precisely define the security properties that prevent the attacks described above and techniques for providing those properties.

II.1.1 Properties

Recall the high-level user goals presented in Section 1.1.2. The first two—Protect Personal Information from Disclosure and Get "Correct" Data—both have to do with guarantees we would like about the data we send and receive over the Internet. Both goals are achieved if we can provide the following three properties:

> **PROPERTY 1: Secrecy**
> Adversaries cannot read application data. (This is sometimes referred to as **confidentiality**.[1])

PROPERTY 2: Entity Authentication.
An endpoint can verify that the other endpoint is truly the entity with which it expects to communicate.

PROPERTY 3: Data Authentication.
Upon receiving a message, an endpoint can verify (1) that it was originated by the other endpoint and (2) it was not modified in flight (this second requirement by itself is sometimes referred to as **data integrity**).

The first two properties, secrecy and entity authentication, are both required to protect users' personal information from disclosure. Secrecy is obvious: if an adversary can directly read sensitive data on-the-wire, e.g., by sniffing public Wi-Fi, clearly personal information is not protected from disclosure. Secrecy alone is not enough, however. For example, even if a user's browser uses an encrypted connection to fetch email, if an adversary can trick it into connecting to the adversary's server instead of the true email server and the user supplies their credentials, then the adversary has gained access to the user's email. Therefore, it is also crucial to establish a secret channel *to the correct server*; this is entity authentication.

The second user goal, accessing the "correct" information, requires both entity and data authentication. Whereas above we used entity authentication to ensure we only *gave* our personal information to trusted entities, here we use it to ensure we only *fetch* information from trusted entities. This must be coupled with data authentication, which certifies that each message the user receives really came from that trusted entity (as opposed to being injected into the network by an attacker) and that it was not tampered with in transit. With both entity and data authentication, the user is assured that (1) they are communicating with the expected, trusted party, (2) each message they receive was sent by that party, and (3) no one else changed the data since it was sent.

II.1.2 Techniques

Properties 1, 2, and 3 can be provided with three well-known techniques, respectively (Figure II.1):

TECHNIQUE 1: Encryption
Encryption provides *secrecy* by preventing anyone without the decryption key from reading the encrypted data.

TECHNIQUE 2: Asymmetric Key Exchange + Public Key Infrastructure (PKI)
Asymmetric key exchange protocols [167], like Diffie-Hellman, allow two parties who have had no prior contact to establish a shared secret. A PKI ties the public key used in that

[1]We use the term *secrecy* instead of *confidentiality* because in many contexts confidentiality implies that one party is safeguarding another party's data by keeping it secret on their behalf. For example, healthcare providers are required to keep medical records confidential. Secrecy is more general and applies well in the context of network communication where both endpoints are equal stakeholders in keeping the data they exchange hidden.

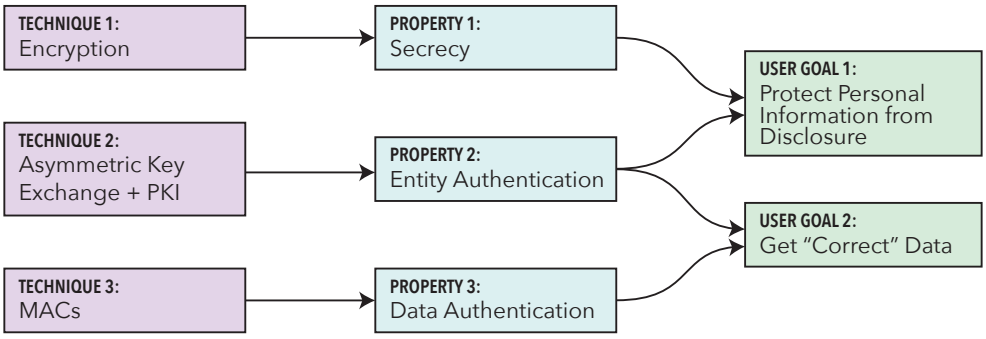


Figure II.1: **Data Protection Techniques.** Properties 1, 2, and 3 from Section II.1.1 are frequently achieved using encryption, asymmetric key exchange + PKI, and message authentication codes (MACs).

key exchange to a particular real-world entity (e.g., a domain name). This makes *entity authentication* possible—users can verify that whomever they just established a key with is the intended party (and then use that key for encryption and/or MAC).

TECHNIQUE 3: Message Authentication Codes (MACs)
MACs are cryptographic checksums that allow you to detect unauthorized changes to a piece of data. A MAC algorithm takes in a message and a symmetric key and produces a token, which is sent along with the original message. The recipient, using the same symmetric key, can use this token to verify that no one tampered with the message. This provides *data authentication*.

II.2 ...Without Giving Up Performance and Functionality

Encryption. Fortunately, there are widely used protocols that implement the techniques described above. The de facto standard is called **Transport Layer Security (TLS)** [94]. TLS uses Encryption to protect data from unauthorized eyes (Secrecy). To establish the symmetric key used for encryption, it uses Asymmetric Key Exchange protocols like Diffie Hellman Key Exchange [95] and relies on a PKI to bind public keys to human-readable entity names (Entity Authentication). And every message sent using TLS is protected with a MAC that allows endpoints to detect whether anyone without the session key modified it (Data Authentication).

There are other variants of TLS. The dominant version today, TLS 1.2, will soon be replaced with TLS 1.3 [196], which promises faster handshakes and fewer options (eliminating weak cipher suites and making the protocol easier to reason about). There is also a variant of TLS, called **Datagram Transport Layer Security (DTLS)** [198], that runs over UDP instead of TCP.

Though TLS is the dominant encryption protocol in today's Internet, there are alternatives. For example, Google's **QUIC** [124] protocol combines the functionality of TCP and TLS into a single protocol with shorter handshakes (like TLS 1.3) and timer-based congestion control. Another example is **TCPcrypt** [63], a TCP extension that encrypts application data at the transport layer by default. Compared to TLS, it (1) imposes less server overhead, (2) offers more authentication options than just certificates, and (3) is more backwards compatible since applications just use TCP

like normal; they do not need an application-level protocol for finding out if the other endpoint supports TLS.

Finally, encryption can also be performed at the network layer using **IPsec** [147].

Middleboxes. In introductory networks courses, students often learn the mantra "smart edge, dumb core." This was one of the guiding design principles in the early days of the Internet: in an effort to curb the complexity of building such an ambitious system, and in recognition of the fact that router technology was weak compared to endpoints, the routers comprising the network itself ("the core") should be as simple as possible. A router's job was just to forward packets towards their destinations and nothing else; any additional functionality was to be implemented on the more powerful endpoints ("the edge").

The world today is completely different. In addition to routers—which are now orders of magnitude more powerful and complex—the network is full of **middleboxes**—devices that process traffic beyond just basic packet forwarding. This includes load balancers, caching proxies, compression proxies, firewalls, virus scanners, intrusion detection systems, and parental content filters. Our focus is on application-layer middleboxes, sometimes called **proxies** or **in-path services**, which we loosely define as middleboxes that access application-layer data (as opposed to, e.g., a firewall operating only on IP and TCP headers).

Middleboxes are everywhere. In a 2012 survey of network operators, networks of all sizes reported having roughly as many middleboxes as they do routers and switches [208]. For web proxies in particular, 14% of Netalyzer sessions show evidence of a proxy [228] and all four major U.S. mobile carriers use proxies—connections to the top 100 Alexa sites are all proxied, with the exception of YouTube on T-Mobile [235].

Encryption + Middleboxes. On their own, encryption and middleboxes are each beneficial: encryption improves security and privacy, and middleboxes improve performance, enhance security, and add other functionality. The problem is, they do not play well together—when traffic is encrypted, middleboxes become blind and can no longer do their jobs. To some this is a good thing. Privacy comes before all else, they argue, and, in fact, blinding middleboxes is often cited as a motivation *for* encryption.

Unfortunately, it is not so simple; we cannot simply turn on HTTPS, close our eyes, and hope for the best. There are several reasons it is important to find a way to use both encryption *and* middleboxes.

Middleboxes are useful. Providing processing and storage in the network has proven to be an effective way to help users, content providers, and network operators alike. For example, techniques such as caching, compression, prefetching, packet pacing, and reformatting improve load times for users [156, 235], reduce data usage for operators and users [37, 103, 175, 190, 232], and reduce energy consumption on the client [97, 130, 175, 190]. Middleboxes can also add functionality not provided by the endpoints, such as virus scanners in enterprises or content filters for children.

Middleboxes can enhance security. Somewhat ironically (considering the arguments against middleboxes tend to revolve around privacy and security), many middleboxes are deployed to *improve* security. If all intrusion detection systems (IDSes), virus scanners, and exfiltration detectors sud-

denly disappeared, this would clearly have a negative impact on security. Worse still, network operators might respond to the prospect of losing these middleboxes by either dropping [181, 187] or transparently intercepting [99] encrypted connections, resulting in a net decrease in security. Finally, users may trust a middlebox more than the application; for example, apps can unexpectedly leak personal information to a server [229], so users may want a middlebox to act as a watchdog.

In-network may be better. First, functions such as caching and packet pacing are inherently more effective in the network [97, 103, 130]. Second, client implementations may be problematic because the client may be untrusted or its software, URL blacklists, virus signatures, etc., may be out-of-date (e.g., only a third of Android users run the latest version of the OS and over half are more than two years out of date [9]). Third, while servers can implement functions such as compression, they may choose not to. A 2014 AT&T study finds that, for the top 500 Alexa sites, only 32% of server responses are compressed; most of these are (already compact) images, but compressing the remainder would yield 11%–13% bandwidth savings [190]. Finally, other functionality, like parental filtering, might depend on local regulations. Having carriers implement these features locally is easier than asking each content provider or client to learn the rules for each country.

They are widely used. In a 2012 survey of network operators, networks of all sizes reported having roughly as many middleboxes as L3 routers [208]. For web proxies in particular, 14% of Netalyzer sessions show evidence of a proxy [228] and all four major U.S. mobile carriers use proxies—connections to the top 100 Alexa sites are all proxied, with the exception of YouTube on T-Mobile [235]. In addition, all actors in the Internet use middleboxes. They are widely deployed in client networks (e.g., enterprise firewalls, cellular networks), and of the three IETF RFCs on using middleboxes with TLS, two are led by operators [156, 159] and one by a content provider [187]. Given this investment, middleboxes are unlikely to go away, so we need a clean, secure way to include them in encrypted sessions.

The Internet is a market-driven ecosystem. The Internet is not a centrally managed monopoly but is a market-driven ecosystem with many actors making independent decisions. For example, while servers can compress data, many only do so selectively [190]. Similarly, content providers may decide not to support device-specific content formatting but instead rely on third party providers, which can be selected by the client or content provider. For functions such as content filtering, clients may decide to pay for the convenience of having a single middlebox provider of their choice, instead of relying on individual content providers. Enterprise networks may similarly decide to outsource functionality [208]. Fundamentally, middleboxes give actors more choices, which leads to competition and innovation.

Do we really need a new protocol? By design, TLS supports secure communication between exactly two parties. Despite this, middleboxes are frequently inserted in TLS sessions, but this has to be done transparently to TLS. Consider an enterprise network that wants to insert a virus scanner in all employee sessions. A common solution is to install a custom root certificate on the client [99, 132, 183]. The middlebox can then create a certificate for itself (purported to be from the intended server) and sign it with the custom root certificate. After the client connects to the middlebox, the middlebox connects to the server and passes the data from one connection to the other. We refer to this as *split TLS* and it gives rise to several problems:

(i) There is no mechanism for authenticating the middlebox. While users can inspect the certificate chain to check who signed the certificate, very few do that or understand the difference. Moreover, even if they do, they have no information about what functions the middlebox performs.

(ii) The client has no security guarantees beyond the first hop. While the connection to the middlebox is encrypted, the client cannot verify that TLS is being used from the middlebox to the server, whether additional middleboxes are present, or (depending on what application-level authentication is used) whether the endpoint of the session is even the intended server. The user needs to completely trust the middlebox, which they may not even know about. A recent study shows that TLS interception proxies used in practice today often reduce security by using weak cipher suites or failing to check server certificates [99].

(iii) Middleboxes get full read/write access to the data stream. Middleboxes can read and modify any data transmitted and received over TLS sessions despite the fact that many middleboxes only need selective access to the data stream (see Table 3.1 in Section 3.2).

Given these problems, it should not be a surprise that users are concerned about (transparent) middleboxes. One could even argue that using TLS in this way is worse than not using TLS at all [67], since clients and servers are under the illusion that they have a secure session, while some of the expected security properties do not actually hold.

In an effort to address the tussle between encryption and middleboxes, the chapters in this part make the following contributions:

- **A study of the impacts of encrypting Web traffic.** In Chapter 2 we present our measurement study on the impact of using TLS, which shows that the loss of middleboxes is the most serious negative effect of switching to HTTPS.
- **Techniques for giving middleboxes access to encrypted data with fine-grained access control.** Chapter 3 presents Multi-Context TLS (mcTLS), our protocol for including middleboxes in secure communication sessions. mcTLS applies the *principle of least privilege* to middleboxes by allowing endpoints to restrict which parts of the data stream middleboxes can read or write.
- **A design space for secure communication protocols that include middleboxes.** Chapter 4 presents a design space for secure multi-entity communication protocols, carefully describing the range of features that such a protocol might have. We also discuss how these features interact and how protocol designers are forced to make tradeoffs.
- **Techniques for securely outsourcing middleboxes.** Chapter 4 also presents Middlebox TLS (mbTLS), another protocol for secure communication including middleboxes that uses trusted computing hardware, like Intel SGX, to protect session data from middlebox infrastructure providers. mbTLS also addresses a number of practical deployability concerns.

II.3 Related Work

There are two high-level approaches for reconciling encryption and middleboxes: (1) design cryptographic primitives that allows middleboxes to *operate directly on* encrypted data, or (2) make a security protocol that allows middleboxes to *decrypt* encrypted data.

II.3.1 Operating on Encrypted Data

The theory community's answer to this problem is **fully homomorphic encryption** [113, 115], which enables an untrusted party to compute arbitrary functions over encrypted data. Unfortunately, these techniques are still orders of magnitude too slow to use in practice [84, 120]. Instead, researchers have developed special-purpose cryptosystems that allow a small set of specific functions to be executed on encrypted data, like search or range queries [38, 49, 215, 238].

Two recent systems bring these techniques to middleboxes. BlindBox [209] introduces a new primitive called DPIEnc, which can be used to encrypt a set of IDS rules and a data stream such that an IDS can match the encrypted rules against the encrypted data stream without ever learning the contents of either. BlindBox has two limitations: (1) it only works for pattern-matching style middleboxes, like IDSes, but cannot handle other middleboxes functions like caching or compression; and (2) it is (currently) too costly to use in practice. Embark [151] is designed specifically to allow network administrators to outsource network-layer middleboxes to a cloud provider without giving the cloud service private information about their network. For example, by encrypting IP headers using a scheme they develop called PrefixMatch, a cloud-based firewall can match encrypted packets to firewall rules without learning what those rules actually are.

II.3.2 Granting Access to Encrypted Data

Several approaches for granting middleboxes access to encrypted communication sessions have been proposed (some of which are already used in practice). Here we briefly summarize the approaches and their limitations.

(1) Custom Root Certificate. This is the approach currently taken by many enterprises. System administrators install a root certificate on employee devices; when an employee starts a connection to a website, the proxy impersonates the site by making a certificate for that site on-demand and signing it with the custom root certificate. It then opens a second TLS connection to the intended destination and forwards data between the client and server. The user agent, unaware there is a proxy at all, thinks it is connecting directly to the server. (In the following chapters, we sometimes refer to this approach as "split TLS.")

Limitations: First, neither the client nor the server knows about the proxy; clearly, then, the proxy is not authenticated to either endpoint. Furthermore, the client cannot tell if the proxy properly authenticated the server or if the proxy–server connection is even encrypted at all. Finally, users are taught bad habits if they install the certificate themselves, and, in either case, most companies probably do not protect their signing keys as well as the CAs do, giving hackers looking to steal data from employee devices an attractive attack vector.

(2) "I'm a proxy" Certificate Flag. A 2014 IETF draft from Ericsson and AT&T proposes using the X.509 Extended Key Usage extension to indicate that a certificate belongs to a proxy [159]. Upon receiving such a certificate during a TLS handshake, the user agent would omit the domain name check (presumably with user permission) and establish a TLS session with the proxy, which would in turn open a connection with the server. Based on user preferences, the user agent might only accept proxy certificates for certain sessions.

Limitations: In this case, the client is aware of the proxy but the server is not. Second, as above, the client cannot tell whether the connection from the proxy to the server is properly authenticated and encrypted.

(3) Pass Session Key Out-of-Band. Another IETF draft, this one from Google, assumes that the client maintains a persistent TLS connection with the proxy and multiplexes multiple sessions over that connection (much how Google's data compression proxy operates). After establishing an end-to-end TLS connection with the server (which the proxy blindly forwards), the client passes the session key to the proxy before transmitting data on the new connection [187]. Again, the user agent can selectively grant the proxy access on a per-connection basis based on user preference.

(4) Ship a Custom Browser. A fourth option is to modify the browser itself to accept certificates from certain trusted proxies. This is the approach Viasat takes for its Exede satellite Internet customers [156], arguing that caching and prefetching are critical on high-latency links.

Limitations: This solution is essentially the same as (1), so it shares the same limitations. In addition, it has the drawback that a custom browser might not be updated quickly, is expensive to develop and maintain, and may be inconvenient for users.

(5) Proxy Server Extension. The most promising approach so far is Cisco's TLS Proxy Server Extension [164]. The proxy receives a ClientHello from the client, establishes a TLS connection with the server, and includes the server's certificate and information about the ciphersuite negotiated for the proxy-server connection in a ProxyInfoExtension appended to the ServerHello it returns to the client. The client can then check both the proxy's and the server's certificate.

Limitations: The client must *completely* trust the middlebox to provide honest information about the server certificate and ciphersuite. The proxy is not necessarily visible to the server.

Other Approaches. An alternative to TLS-based techniques is an extension to IPsec that allows portions of the payload to be encrypted/authenticated between the two end-points of a security association and leaves the remainder in the clear [144]. The authors target this architecture for securely enabling intermediary-based services for wireless mobile users. This solution leaves data for middleboxes completely unencrypted.

Another alternative to a transport-layer protocol, like TLS, is supporting trusted intermediaries at the network layer. The Delegation-Oriented Architecture (DOA) [225] and Named Data Networking (NDN) [138] do this with their own security mechanisms and properties. Unlike the techniques above, these approaches cannot be immediately deployed.

Chapter 2 The Cost of Ubiquitous Encryption

Increased user concern over online security and privacy (specifically, User Goals 1 and 2) has led to widespread adoption of TLS, the protocol for encrypted, authenticated communication we introduced in Section II.2. However, as with any security solution, it does not come for free. TLS may introduce overhead in terms of infrastructure costs, communication latency, data usage, and energy consumption. Moreover, given the opaqueness of the encrypted communication, any middleboxes requiring visibility into application layer content, such as caches and virus scanners, become ineffective.

This chapter sheds light on these costs by studying HTTPS, the secure version of HTTP (which uses TLS). First, using datasets collected from large ISPs, we examine the accelerating adoption of HTTPS over the last three years. Second, we quantify the direct and indirect costs of this evolution. Our results show that, while deploying TLS has some direct performance impact, these effects can largely be "engineered away." The far bigger concern is the loss of middleboxes, which we show can measurable negative impact on (among other things) data usage, energy usage, and security. This result motivates the work we present in Chapter 3 and Chapter 4.

2.1 Introduction

The HyperText Transfer Protocol (HTTP) was first introduced in the early '90s. Since then, the Internet has changed significantly, becoming a vital infrastructure for communication, commerce, education, and information access. HTTPS, the secure version of HTTP, runs HTTP on top of TLS[94]. While originally geared toward services that require data confidentiality or authentication between client and server, like online banking or e-commerce, the increasing personalization of the web has led to a number of other services adopting HTTPS, such as GMail, Facebook, and even YouTube. Furthermore, all major browsers require the use of TLS for all HTTP/2 [135] connections, mirroring a fundamental design decision of SPDY [220], which was used as the starting point for HTTP 2.0.

Given users' growing concerns about security and privacy on the Internet, adopting encryption by default in all HTTP communication sounds like a good idea. However, security always comes at a cost, and HTTPS is no different (graphically depicted in Figure 2.1). In this chapter, we aim to categorize and quantify the cost of the "S" in HTTPS.

First, we look at the way HTTPS adoption has evolved over the past three years. Such an

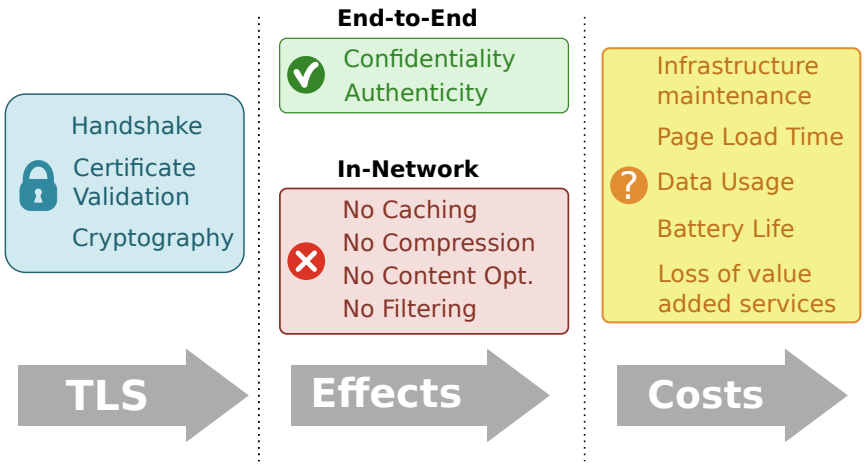


Figure 2.1: **Impact of TLS.** The mechanisms comprising TLS, their effect on the netowrk, and the potential costs of those effects.

analysis is important because, besides quantifying trends, it sheds light on the cost of deploying HTTPS for web services, a cost that seems to be diminishing: 50% of web traffic flows today are secure, including, for the first time, large content (e.g., 50% of YouTube streaming flows are over HTTPS).

Second, we study how TLS impacts latency, data consumption, and battery life for clients. HTTPS requires an additional handshake between the client and the server in addition to the added computational cost of cryptographic operations. We study how significant these costs are for fiber, Wi-Fi, and 3G connections.

Lastly, while encryption provides a clear value to the end user in terms of confidentiality and authentication, it could have implications that are harder to assess. Over the past 15 years, an increasing number of network functionalities have been performed by transparent and explicit middleboxes, aiming to reduce the amount of backbone traffic, compress content before transmission on expensive wireless links, filter inappropriate/undesired content, and protect users and organizations from security threats. These boxes suddenly become blind in the presence of encryption. We show that there are clear cases where losing such functionality may not only harm network efficiency, but also increase latency by more than 50% and consume up to 30% more energy for 3G mobile devices.

Using three different datasets, captured in residential and cellular networks, as well as controlled experiments, our work shows that: (i) HTTPS usage is increasing despite potential deployment costs, (ii) HTTPS has a perceptible impact on clients in terms of latency, (iii) its data overhead seems to be limited, (iv) it could lead to significantly increased battery consumption for large objects.

This creates a complex tradeoff that depends on many factors, including context and personal preference. While specific workloads will always be served through HTTPS (e.g., financial transactions), there are a number of applications where encryption could be optional or the service could be assisted by *trusted proxies*, which bring back the benefits of middleboxes, as described in a recent IETF proposal [159]. How to best manage this tradeoff is an open question that challenges the research community.

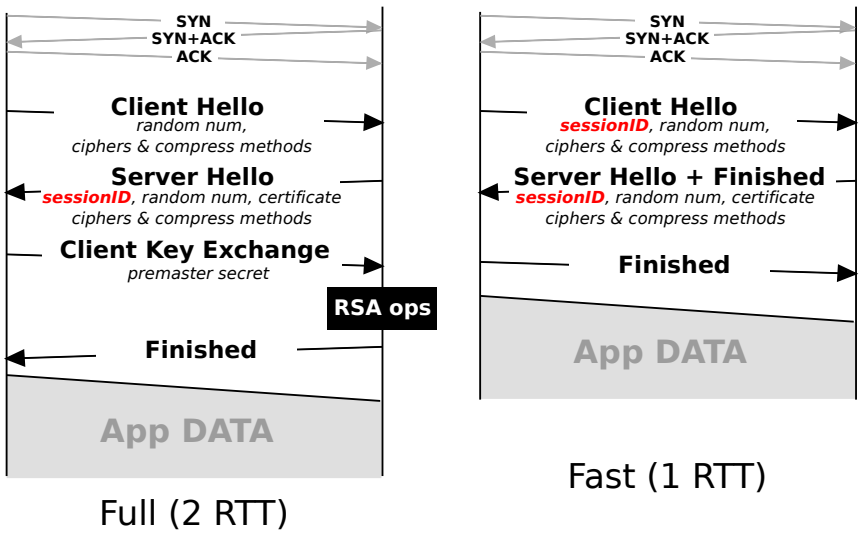


Figure 2.2: **TLS Handshake.** If the server supports it, subsequent connections from the same client to the same server can use a 1 RTT version of the handshake.

In the context of this thesis, our results show that the the biggest potential cost, though hard to quantify, is the loss of middleboxes, which we show can measurable negative impact on (among other things) data usage, energy usage, and security. This result motivates the work we present in Chapter 3 and Chapter 4.

2.2 HTTPS Overview

SSL/TLS is the standard protocol for providing authenticity and confidentiality on top of TCP connections. Today, it is used to provide a secure version of traditional protocols (e.g., IMAP, SMTP, XMPP, etc.); in particular, the usage of HTTP over TLS is commonly known as HTTPS.

Each TLS connection begins with a *handshake* between the server and the client. In this handshake, the Public Key Infrastructure (PKI) suite is used to authenticate the server (and sometimes the client) and to generate cryptographic keys to create a secure channel for data transmission.

Figure 2.2 (left) sketches the steps in a *full* TLS negotiation. In this scenario, the client and the server incur different costs. On the server side, the primary cost is computing the session key. This involves complex public key cryptography operations (typically RSA), limiting the number of connections per second the server can support [42, 72, 81]. For clients, the major cost is latency. This depends on (i) server performance, (ii) the distance between client and server since a *full* negotiation requires two RTTs (three including the TCP handshake), and (iii) if the application chooses to do so, the latency to verify the server's certificate with the PKI (e.g., an OSCP/CRL check).

Unsurprisingly, a few optimizations have been proposed to reduce handshake costs. Hardware accelerators, GPU architectures [140], or "rebalancing" the RSA computations [72] can easily boost server performance by a factor of 10. Also, the TLS standard provides a *fast* negotiation mechanism,

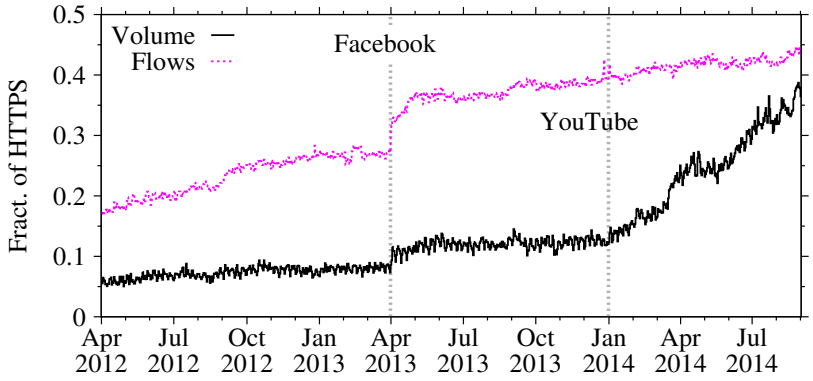


Figure 2.3: **HTTPS Adoption over Time.** Evolution of HTTPS volume and flow shares over 2.5 years. Results from Res-ISP dataset. Vertical lines show the transition to HTTPS for Facebook and YouTube.

shown in Figure 2.2 (right). In this case a `SessionID` is used to retrieve a previously negotiated session key, (i) avoiding the cost of creating a new session key and (ii) reducing the handshake to 1 RTT (2 including the TCP handshake). Note that the adoption of fast negotiation is controlled by the server; as we see in Section 2.4, only some services deploy it.

2.3 HTTPS Usage Trends

A common belief is that deploying HTTPS increases infrastructure costs (to accommodate the resulting computational, memory, and network overhead) in addition to the cost of certificates (up to $1,999/year each[1]). Thus, one would expect services to carefully deploy HTTPS only when needed. To test this, we examine recent HTTPS usage trends. We collected per-flow logs from a vantage point monitoring the traffic of about 25,000 residential ADSL customers of a major European residential ISP ("Res-ISP"). The vantage point runs Tstat [109], which implements a classifier supporting both HTTP and TLS identification. For TLS traffic, Tstat parses the `ClientHello` and `ServerHello` TLS handshake messages to extract (i) Server Name Indication (SNI), i.e., the hostname to which the client is attempting to connect, and (ii) Subject Common Name (SCN) carried in the server certificate, i.e., the name the server itself presents. Tstat is also able to identify the presence of SPDY in the TLS connections. In the following, we use this rich ISP dataset to characterize the evolution of HTTPS usage. At the time of writing, HTTPS and HTTP combined represent 75% of all TCP traffic (by volume) in Res-ISP.

Figure 2.3 reports the evolution of the HTTPS traffic share from April 2012 to September 2014. Both volume and flow shares are shown. The growth of HTTPS adoption is striking, with the HTTPS flow share more than doubling in two years. In September 2014, 44.3% of web connections already use HTTPS.[2] The sharp bump in April 2013 is due to Facebook enabling HTTPS by default for all users [194].

[1] https://www.symantec.com/page.jsp?id=compare-ssl-certificates

[2] Curiously, only 5.5% of flows successfully negotiated SPDY, despite 55% of clients offering the option. This highlights that SPDY is only supported by a handful of (popular) services, namely Google and Facebook.

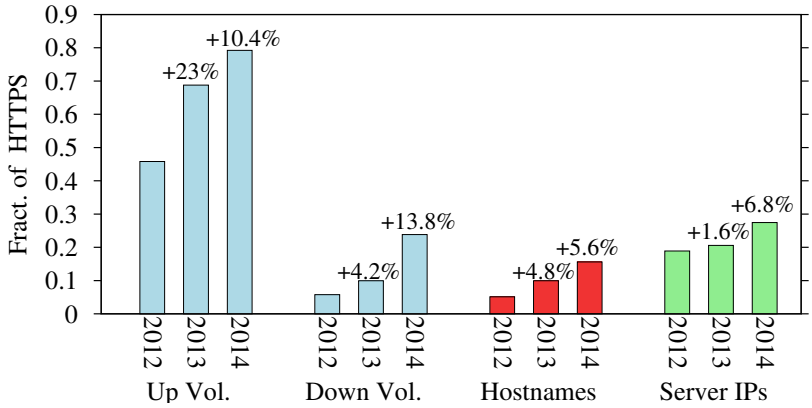


Figure 2.4: **HTTPS Usage by Volume, Domain, and IP.** Comparing HTTPS shares over three one-week periods in the Res-ISP dataset. Percentages highlight year-to-year growth.

Looking at volume, we see a much slower growth. Intuitively, one would expect that most HTTPS flows carry small, privacy-sensitive objects. We see this until January 2014 when YouTube began delivering video content over HTTPS, clearly increasing the HTTPS volume share. As of September 2014, as much as 50% of YouTube's aggregate traffic volume is carried over HTTPS.

Figure 2.4 further details HTTPS trends with respect to upload volume, download volume, number of hostnames, and number of server IP addresses. We compare the first week of April in 2012, 2013 and 2014 (results are consistent for other weeks). Percentages show the year-to-year increase. The growth of HTTPS is again evident. For instance, HTTPS accounts for 80% of the upload volume in 2014; it was only 45.7% in 2012. This reflects the fact privacy-sensitive information tends to be uploaded using HTTPS more and more. Interestingly, the fraction of data downloaded using HTTPS is smaller than the fraction uploaded. However, YouTube's shift to HTTPS in 2014 dramatically changed the landscape: HTTPS download volume more than doubled compared to 2013. Figure 2.4 also highlights a constant year-by-year increase for both the fraction of hostnames and server IP addresses accessed using TLS. Interestingly, 72% of the TLS hostnames are accessed exclusively over TLS in 2014.

These results clearly show that, despite the perceived costs, services are rapidly deploying HTTPS. This shift is undoubtedly related to the recent attention toward guaranteeing end-users privacy. However, this may also be an indication of increasingly manageable infrastructural costs. This is consistent with the report from the GMail team after their switch to HTTPS: "On our production frontend machines, SSL/TLS accounts for less than 1% of the CPU load, less than 10KB of memory per connection and less than 2% of network overhead." [152] Reports from Facebook are similar [51].

Takeaway: *HTTPS accounts for 50% of all HTTP connections and is no longer used solely for small objects, suggesting that the cost of deployment is justifiable and manageable for many services.*

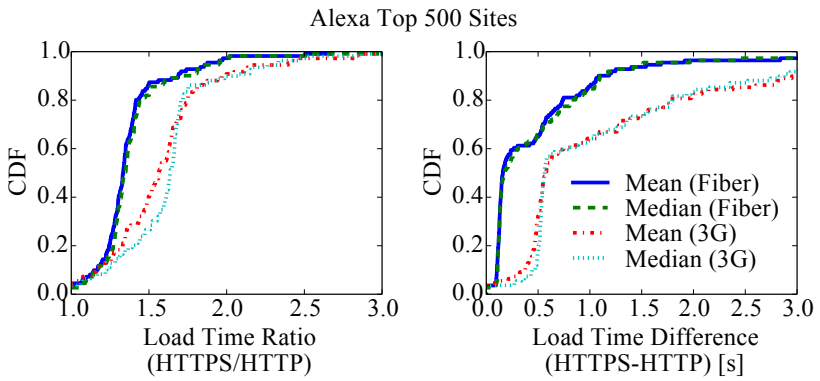


Figure 2.5: **Page Load Time.** Webpage load time inflation for the Alexa top 500.

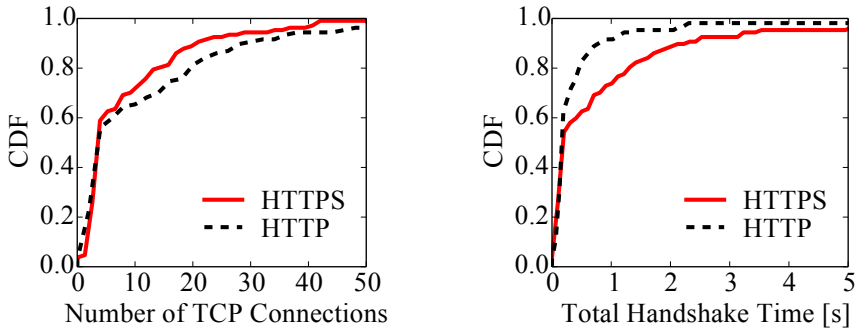


Figure 2.6: **Handshakes.** Number of TCP handshakes (left) and total handshake time (right) for the Alexa top 500 (fiber).

2.4 Page Load Time

We now investigate how much HTTPS affects load time, an important quality-of-experience metric for web browsing.

Overall Page Load Time. First we quantify the HTTPS page load time overhead through active experiments. We load each of the top 500 Alexa sites 20 times, first using HTTP and then using HTTPS. For the download, we use PhantomJS [21], a fully-fledged headless browser running on a Linux PC. From each run, we extract the average and median load times. The test PC is connected first using a 3G USB modem and then via fiber, two typical real-world environments. The local cache is cleared between page loads.

Results are reported in Figure 2.5. Plots show the Cumulative Distribution Function (CDF) of the ratio of HTTPS to HTTP page load time (left) and the absolute difference (right). The benchmark shows that using HTTPS significantly increases load time. This is especially evident on 3G, where the extra latency is larger than 500 ms for about 90% of websites, and 25% of the pages suffer an increase of at least 1.2 s (i.e., an inflation factor larger than 1.5x). On fiber, the extra latency is smaller; still, for 40% of the pages, HTTPS adds more than 500 ms (1.3x).

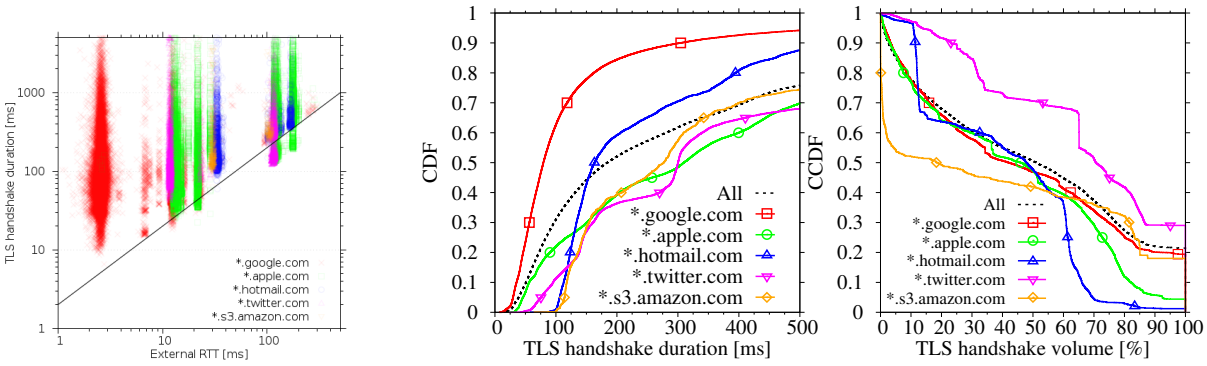


Figure 2.7: **TLS Handshake Costs.** Scatter plot of the TLS handshake duration with respect to server distance (left). TLS handshake duration CDF (center). Ratio of TLS handshake bytes with respect to total TCP connection bytes CCDF (right).

We also captured an HTTP Archive (HAR) for each page, which contains statistics about the individual connections used to load the page. Figure 2.6 (left) shows 40% of the sites open fewer TCP connections when loaded over HTTPS compared to HTTP. Even so, Figure 2.6 (right) shows nearly half of the sites spend *more time* establishing those connections; the increased time per handshake is caused primarily by TLS negotiation overhead (see below). By examining the HARs, we see that many of the sites using fewer connections serve fewer objects (1–3 fewer on average, but in some cases upwards of 100 fewer) and do so from fewer hosts (typically 1–2 fewer, but in some cases up to 40 fewer). We suspect these are intentional changes to the HTTPS version of the site designed to avoid costly TLS handshakes.

TLS Handshake delay cost: Webpage load time has been an active area of research [102, 104, 123, 136, 224, 226, 227]; understanding the exact cause(s) of the inflation noted above is quite complex and out of scope for this work. However, it is still interesting to understand whether the overall page latency is primarily affected by network latency or by protocol overhead. To better understand this, we modified Tstat to extract the (i) duration and (ii) number of bytes of each TLS handshake from a one-hour pcap trace collected on April 3rd 2014 from Res-ISP. About 1 million TLS flows are present.

Figure 2.7 (left) shows a scatter plot of the TLS handshake duration with respect to the minimum external RTT (i.e., the RTT between the vantage point and the remote server[3]). The external RTT is reasonably representative of the distance to the remote server. For this analysis we selected popular services as representative cases (we classify based on the Server Name Indication in the `ClientHello`).

First, all services exhibit vertical clusters of points, which likely reflect the different data centers offering that service. For instance, when the external RTT is larger than 100 ms, the server is outside Europe. More interestingly, no matter how close the servers are, all clusters contain samples with very high TLS handshake duration, i.e., up to several seconds. To better capture this effect, Figure 2.7 (center) reports the CDF of the TLS handshake duration for individual services and

[3]Tstat extracts RTT per-flow statistics monitoring the time elapsed between TCP data segments and the corresponding TCP ACK.

for the traffic aggregate (black dotted line). Google services (which are also the closest) exhibit the smallest TLS negotiation delay, though 10% of measurements are more than 300 ms. Since a full TLS handshake requires at least 2 RTT (1 RTT in case of `SessionID` reuse), services handled by U.S. servers (e.g., Hotmail, Twitter, Amazon S3) experience huge extra costs. For instance, for Twitter, negotiation takes over 300 ms for more than 50% of the HTTPS connections. In general, 5% of requests experience a handshake at least 10 times longer than the RTT. This might be due to client or server overhead, network congestion, or a slow OCSP check.

Looking closer, we find that 4% of clients experience at least one connection with a TLS handshake duration higher than 300 ms. For such connections, 50% (75%) have an internal RTT (i.e., the RTT between the vantage point and the end-user device) of 51 ms (97 ms). The same holds true with less conservative thresholds (e.g., 1 second). This demonstrates that even clients with good network connectivity can still significantly suffer from TLS handshake overhead. TLS fast negotiation can help to reduce the handshake latency, but we find this being used in only 30% of the connections. We speculate this represents a lower bound, but, unfortunately, based on available data, we cannot assess the achievable upper bound obtained from a wider adoption of TLS fast negotiation.

Takeaway: *The extra latency introduced by HTTPS is not negligible, especially in a world where one second could cost 1.6 billion in sales [100].*

2.5 Data Usage

HTTPS also impacts the volume of data consumed due to (i) the size TLS handshake and (ii) the inability to utilize in-network caches and compression proxies.

TLS Handshake Data Cost. The impact of the TLS handshake overhead depends on how much use the connection sees; the more data transferred, the lower the relative cost of the negotiation packets. Figure 2.7 (right) reports the Complementary Cumulative Distribution Function (CCDF) of the ratio between TLS handshake size and total bytes carried in the TCP connection. Results refer to a peak hour in April 2014 for the Res-ISP dataset and are consistent with other time periods. We see that many TLS connections are not heavily used. In fact, for 50% of them, the handshake represents more than 42% of the total data exchanged. However, some services, like those running on Amazon S3, do actually use connections efficiently, reducing the impact of the negotiation cost. Some services also try to mask negotiation latency by "pre-opening" connections before they actually need to send data. In this case, the negotiation overhead is 100% if the connection is never used. This is captured in the rightmost part of Figure 2.7 (right), which also highlights how this optimization is heavily used by Google, Amazon S3, and Twitter, but is not by Hotmail and Apple services. Despite all this variability, the average TLS negotiation overhead amounts to 5% of the total volume in this dataset.

In-Network Proxies. HTTPS prevents in-network content optimizations, like proxies that perform compression and caching. To evaluate the impact of this loss, we analyze logs from two production HTTP proxies for mobile networks: Transp-Proxy and OptIn-Proxy. Transp-Proxy

refers to a transparent proxy in a major European mobile carrier serving more than 20 million subscribers. OptIn-Proxy, on the other hand, is an explicit proxy serving 2000 mobile subscribers daily in a major European country. For Transp-Proxy, we analyze the past two years of traffic and for OptIn-Proxy we consider a week-long trace from May 2014.

Caching (ISP Savings): An ISP can save upstream bandwidth by serving static content from its own transparent cache. In the Transp-Proxy dataset, the average cache hit ratio over the past two years was 14.9% (15.6% of the total data volume), amounting to savings of 2 TB per day for a single proxy instance serving 3 million subscribers. For OptIn-Proxy, we see daily savings of 1.3 GB, which, if scaled up to the Transp-Proxy population, matches the observed Transp-Proxy savings.

We also witnessed a decrease in cache efficiency: the cache hit ratio of Transp-Proxy dropped from 16.8% two years ago to 13.2% in June 2014. Based on our analysis, it is not easy to conclude how much of the decreasing effectiveness of caching is related to the adoption of HTTPS and how much is caused by the increased personalization of web traffic. Either way, savings of this size can be still be significant to network operators; such savings will be totally eliminated if content delivery moves entirely to HTTPS.

Compression (Users Savings): Before returning content to users, web proxies typically apply lossless (e.g., gzip) compression to objects and, in more aggressive settings, even scale or re-encode images. This functionality is particularly helpful in cellular networks where the capacity is limited and where users often have restrictive data allowances. The Transp-Proxy trace shows a compression ratio of 28.5% (i.e., the last-mile of the network and the users save one-third of the original data size). In terms of average volume, this amounts to only 2.1 MB per user per day (on average a mobile user downloads less than 10 MB per day). For heavy users, though, this may translate to significant savings (e.g., more than 300 MB per month).

Takeaway: *Most users are unlikely to notice significant jumps in data usage due to loss of compression, but ISPs stand to see a large increase in upstream traffic due to loss of caching.*

2.6 Battery Life

HTTPS has the potential to negatively impact battery life (particularly on mobile devices) due to (i) the extra CPU time required for the cryptographic operations and (ii) increased radio uptime due to longer downloads.

Synthetic Content. To measure the raw energy overhead of HTTPS, we instrumented a Galaxy S II with a power meter that samples the current drawn by the phone every 200 μs. We used the test phone to download synthetic objects over 3G and Wi-Fi (i.e., an access point connected to a fiber link). Objects range in size from 1 kB to 1 MB and are hosted on a web server under our control. Objects are downloaded 100 times each over HTTP and HTTPS using `curl` (compiled for Android). We configured the server to deliver traffic avoiding any proxy cache along the path.[4] During our tests, the screen was on at its minimum brightness.

[4]The 3G carrier used in the experiment runs transparent proxies acting on traffic to port 80. By configuring the webserver to listen on a different port, we bypass the cache.

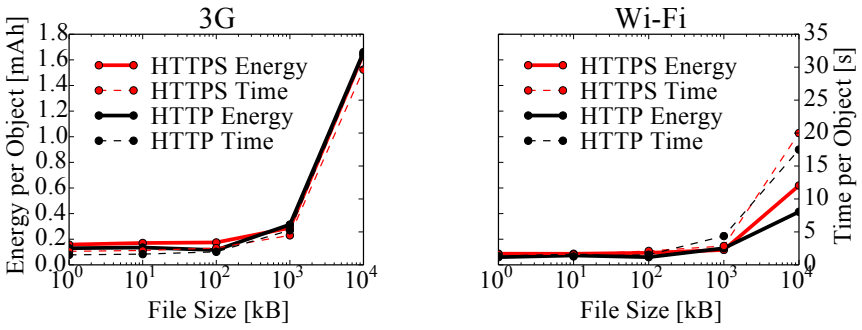


Figure 2.8: **Energy Consumption.** Energy consumption on 3G (left) & Wi-Fi (right).

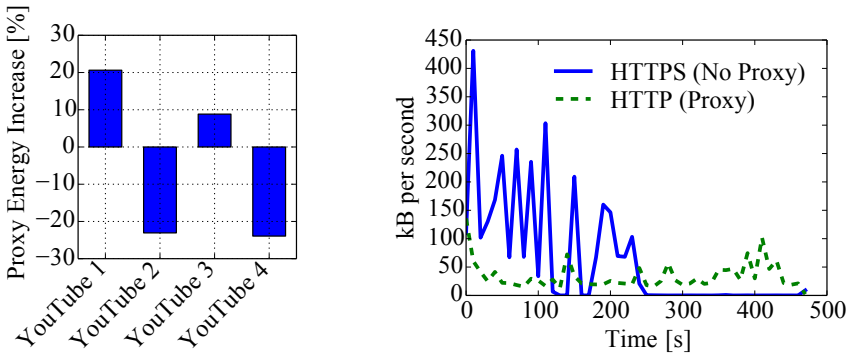


Figure 2.9: **Energy Consumption During Video Streaming.** Comparing YouTube video playback over HTTP (with proxy) and HTTPS (without proxy): energy consumption increase when using HTTP+proxy (left) and download rate over time for one video (right). Results for 3G.

Figure 2.8 shows both average time (right y-axis) and energy (left y-axis) to complete each download. It is immediately clear that energy consumption is strongly correlated to download time; this is not surprising, as leaving the radio powered up is costly. (We also see a slight overhead for large objects over HTTPS on Wi-Fi but not on 3G, but we were unable to precisely determine the cause. The difference is less than one standard deviation.) The key takeaway here is, download time aside, we do not see a noticeable overhead due to cryptographic operations.

	Total Energy (mAh)		**Avg. Current (mA)**	
	3G	*Wi-Fi*	*3G*	*Wi-Fi*
HTTP	210.8	175.7	633	520
HTTPS	217.7	178.0	653	531

Table 2.1: **Energy Consumption During Web Browsing.** Energy consumed loading CNN homepage.

Real Content. We complement the previous analysis by loading real content. We mirror the CNN homepage on our controlled webserver and download it 50 times using Chrome for Android over HTTP and HTTPS (enforcing 20 seconds of wait between consecutive downloads). Results are listed in Table 2.1 (numbers presented are cumulative for all 50 loads). As in the previous benchmark, HTTPS tests do not show an appreciable increase in energy costs.

In a second experiment, we play four 5- to 12-minute YouTube videos. Since the YouTube app does not deliver video content over HTTPS for mobile devices (nor does the YouTube mobile site), we first force the phone to load the desktop version of the YouTube portal. Over Wi-Fi, there was no difference; on 3G, on the other hand, our network's Web proxy significantly impacted the HTTP results. For two videos, playback over HTTP (with proxy) consumes nearly 25% less energy than over HTTPS (without proxy); for the other two, 10%–20% more (Figure 2.9 left).

The differences are caused by two distinct proxy behaviors. First, the proxy throttles the download rate (Figure 2.9 right) to reduce congestion and avoid wasting bandwidth if the user abandons the video. Without the proxy (HTTPS), the whole video loads immediately and the radio sleeps while it plays. With the proxy (HTTP), the download is slow and steady, lasting the duration of the video. Without the opportunity to sleep, playback over HTTP consumes more energy. Second, the proxy injects javascript into the pages it returns, which, among other things, rewrites the URLs sent to YouTube to request encodings and qualities more appropriate for mobile devices. For YouTube 2 and YouTube 4, the player requests the content in the `webm` format, for which our phone does not have hardware decoding support; the proxy changes `webm` to `mp4`, which our phone can decode in hardware. The benefit of hardware decoding outweighed the cost of radio uptime.

Of course, these numbers should be taken with a grain of salt, since using the desktop version of YouTube on a phone is unrealistic (but they are still relevant to PC users connecting via USB modem or tethering). We played the same four videos with YouTube's mobile portal in addition to two new videos from Vimeo's mobile site and verified that the mobile players request `mp4` from the start, so the proxy does not help decrease decoding costs. The mobile video was still throttled.

Stepping back, we see these results as concrete examples of a proxy helping and a proxy hurting end users, suggesting that (1) operators should think carefully about how they configure middleboxes and (2) the community should think carefully about shutting them out by switching to HTTPS by default.

Takeaway: *HTTPS' cryptographic operations have almost no impact on energy costs, but the loss of proxies can significantly impact battery life (positively and negatively).*

2.7 Loss of Middleboxes

In Section II.2 we introduced middleboxes—devices in the network that process traffic beyond basic packet forwarding. Encrypting traffic blinds middleboxes; they can no longer do their jobs when all they see is encrypted bits. Though it is not always easy to quantify, the loss of middleboxes is almost surely the biggest cost caused by ubiquitous encryption. In this section, we discuss some of these costs.

Middleboxes are widely used for the simple reason that adding processing and storage in the network has proven to be an effective way to help users, content providers, and network operators by,

e.g., improving performance, reducing cost, or providing functionality not offered by the endpoints. Some examples:

Load Time. Web proxies can decrease page load time with caching and compression. In mobile networks, image compression can decrease page load time by 5X [235] and in extreme scenarios, like satellite links, actively prefetching and pushing content can yield 4X improvements [156].

Data Usage. Google boasts its data compression proxy can reduce the size of a web page up to 50% and device-specific image resizing can shrink images up to 80% [10]. We saw earlier in Section 2.5 that text compression and image resizing reduce data consumption by nearly a third and 16 TB are served from the cache daily, saving upstream bandwidth and reducing traffic on the air. TCP-level redundancy elimination (RE) can go further, saving 30%–40% [232].

Energy Consumption. As we saw in Section 2.6, selecting alternate video encodings for certain YouTube videos can decrease energy consumption by 10%–20% on some devices. Scheduling packet delivery can allow radios to sleep up to 70% of object transfer time [97].

Security. Ubiquitous encryption will render all deep packet inspection (DPI) boxes ineffective. The advantage that in-network DPIs have is the ability to observe the traffic of multiple clients at the same time to draw inferences, while access to application layer content allows them to block threats by searching for pre-defined signatures (e.g., a known malware binary). Unsophisticated DDoS attacks may still be detectable through statistical analysis of the HTTPS traffic, but application layer fingerprinting will have to be pushed to the client. Another example is virus scanning, which many enterprise networks perform on all inbound traffic to protect employee machines.

Additional Functionality. Middleboxes can add other kinds of functionality not provided by the endpoints. For instance, a number of telecommunications providers today provide parental filtering through the use of explicit blacklists, such as the Internet Watch Foundation[5] list. Through direct communication with IWF, we found out that only 5% of their current blacklist is pure domains or sub-domains that could still be blocked in the presence of HTTPS. To maintain full functionality, the IWF list would have to again be moved to the client, where one can still observe the complete URL being accessed.

Other opt-in services offered by some providers are similarly affected, like content prioritization (e.g., postponing ad delivery) or blocking tracking cookies. (Interestingly, losing the ability to block tracking cookies hurts privacy, which is one of the goals of using TLS to begin with.)

Takeaway: *Though difficult to quantify, the loss of in-network services is potentially substantial; some of that functionality could be equally well performed on the client, while some may require a total rethink, like DPI-based Intrusion Prevention Systems (IPSes).*

[5] https://www.iwf.org.uk

2.8 Conclusion

Motivated by increased awareness of online privacy, the use of HTTPS has increased in recent years. Our measurements reveal a striking ongoing technology shift, indirectly suggesting that the infrastructural cost of HTTPS is decreasing. However, HTTPS can add direct and noticeable protocol-related performance costs, e.g., significantly increasing latency, critical in mobile networks.

More interesting, though more difficult to fully understand, are the *indirect* consequences of the HTTPS: most in-network services simply cannot function on encrypted data. For example, we see that the loss of caching could cost providers an extra 2 TB of upstream data per day and could mean increases in energy consumption upwards of 30% for end users in certain tases. Moreover, many other value-added services, like parental controls or virus scanning, are similarly affected, though the extent of the impact of these "lost opportunities" is not clear.

What *is* clear is this: the "S" is here to stay, and the network community needs to work to mitigate the negative repercussions of ubiquitous encryption. To this end, we see two parallel avenues of future work: first, low-level protocol enhancements to shrink the performance gap, like Google's ongoing efforts to achieve "0-RTT" handshakes.[6] Second, to restore in-network middlebox functionality to HTTPS sessions, we expect to see trusted middleboxes [159] become an important part of the Internet ecosystem. The next two chapters of this thesis develop protocols for safely including trusted middleboxes in secure communication sessions.

[6] http://blog.chromium.org/2013/06/experimenting-with-quic.html

Chapter 3 Access Control for Middleboxes in TLS

In Chapter 2, we saw that a significant fraction of Internet traffic is now encrypted. However, TLS assumes that all functionality resides at the endpoints, making it impossible to use middleboxes that optimize network resource usage, improve user experience, and protect clients and servers from security threats. Furthermore, in Section II.2 we saw that re-introducing in-network functionality into TLS sessions today is done through hacks, often weakening overall security.

In this chapter we introduce **Multi-Context TLS (mcTLS)**, which extends TLS to support middleboxes. mcTLS breaks the current "all-or-nothing" security model by allowing endpoints and content providers to explicitly introduce middleboxes in secure end-to-end sessions. mcTLS' key feature is *access control*: the endpoints can control which parts of the data they can read or write.

We evaluate a prototype mcTLS implementation in both controlled and "live" experiments, showing that its benefits come at the cost of minimal overhead. More importantly, we show that mcTLS can be incrementally deployed and requires only small changes to client, server, and middlebox software.

3.1 Introduction

As we saw in Chapter 2, 40% of all HTTP flows use TLS and this number is estimated to grow by 40% every 6 months [180]. This is good news for privacy. However, TLS makes a fundamental assumption: all functionality must reside at the endpoints. In reality, as we introduce in Section II.2, Internet sessions are augmented by middleboxes providing services like intrusion detection, caching, parental filtering, content optimization (e.g., compression, transcoding), or compliance to corporate practices in enterprise environments. These functional units, often referred to as *middleboxes*, offer many benefits to users, content providers, and network operators, as evidenced by their widespread deployment in today's Internet.

In Section II.2, we describe the "split TLS" approach used today for including middleboxes in TLS sessions and argue why this approach broken from a security perspective. Recently, industrial efforts—e.g., one by Ericsson and AT&T [159] and one by Google [187] offer improvements, but as we discuss in Section 3.6, still lack some important security properties.

In this chapter we present **Multi-Context TLS (mcTLS)**, a protocol that builds on top of TLS to allow endpoints to explicitly and securely include middleboxes with complete visibility and control.

The key feature of mcTLS is *access control*: the endpoints can control which parts of the data each middlebox can read or write. This is based on the observation that most common middleboxes do not need full read/write access to *all* session data (see Table 3.1); most can successfully do their jobs with fewer permissions—e.g., a parental filter only needs read-only access to HTTP GET headers so it can see the URL being fetched and decide whether to block the connection. mcTLS applies the *principle of least privilege* [200] to middleboxes by allowing endpoints to grant them the minimum level of access they need to do their jobs and nothing more.

In summary, mcTLS:

1. provides endpoints explicit knowledge and control over *which* middleboxes are part of the session.
2. allows users and content providers to *dynamically* choose which portions of content are exposed to in-network services (e.g., HTTP headers vs. content).
3. protects the authenticity and integrity of data while still enabling modifications by selected middleboxes by separating read and write permissions.
4. is incrementally deployable.

We implemented mcTLS in the OpenSSL library. Our evaluation shows that mcTLS has negligible impact on page load time or data overhead for loading the top 500 Alexa sites and incorporating mcTLS into applications is relatively easy in many cases.

Our contributions are as follows:

1. a practical extension to TLS that explicitly introduces trusted in-network elements into secure sessions with the minimum level of access they need.
2. an efficient fine-grained access control mechanism which we show comes at very low cost.
3. strategies for using mcTLS to address concrete, relevant use cases, many of which can immediately benefit applications with little effort using mcTLS's most basic configuration.
4. a prototype implementation of mcTLS tested in controlled and live environments (our implementation is available online [1]).

3.2 Multi-Context TLS (mcTLS)

This section presents the design of multi-context TLS (mcTLS), which augments TLS with the ability to securely introduce trusted middleboxes. Middleboxes are trusted in the sense that they have to be inserted explicitly by either the client or the server, at both endpoints' consent. We first summarize our design requirements, then introduce the key ideas, and finally describe the key components of the protocol. A more detailed description of mcTLS is available online [1].

3.2.1 Protocol Requirements

First, we require mcTLS to maintain the properties provided by TLS (extended to apply to middleboxes):

	Request		Response	
	Headers	*Body*	*Headers*	*Body*
Cache	○		●	●
Compression			●	●
Load Balancer	○			
IDS	○	○	○	○
Parental Filter	○			
Tracker Blocker	●		●	
Packet Pacer			○	
WAN Optimizer	○	○	○	○

(● = read/write; ○ = read-only)

Table 3.1: **Middlebox Permissions.** Examples of app-layer middleboxes and the permissions they need for HTTP. No middlebox needs read/write access to all of the data.

R1: Entity Authentication. Endpoints should be able to authenticate each other and all middleboxes. Similar to TLS usage today, we expect that clients will authenticate all entities in the session, but servers may prefer not to (e.g., to reduce overhead).

R2: Data Secrecy. Only the endpoints and trusted middleboxes can read or write the data.

R3: Data Integrity & Authentication. All members of the session must be able to detect in-flight modifications by unauthorized third parties, and endpoints must be able to check whether the data was originated by the other endpoint (vs. having been modified by a trusted middlebox).

Second, the introduction of middleboxes brings with it two entirely new requirements:

R4: Explicit Control & Visibility. The protocol must ensure that trusted middleboxes are added to the session at the consent of both endpoints. Endpoints must always be able to see all trusted middleboxes in the session.

R5: Least Privilege. In keeping with the principle of least privilege [200], middleboxes should be given the minimum level of access required to do their jobs [158, 172]. Middleboxes should have access only to the portion of the data stream relevant to their function; if that function does not require modifying the data, access should be read-only. Many common middleboxes do not need full read/write access to the entire data stream (see Table 3.1).

Finally, our protocol should meet all five requirements without substantial overhead, e.g., in terms of latency, data usage, computation, connection state, burden on users or administrators, etc.

3.2.2 Threat Model

A successfully negotiated mcTLS session meets the above requirements in the face of computationally bounded network attackers that can intercept, alter, drop, or insert packets during any phase

of the session. Like TLS, mcTLS does not prevent denial of service.

We assume that all participants in an mcTLS session execute the protocol correctly and do not share information out-of-band. For example, the client could share keys with a middlebox not approved by the server, or middleboxes could collude to escalate their permissions. We do not consider such attacks because no protocol (including TLS) can prevent a party from sharing session keys out-of-band. Furthermore, such attacks are unlikely since at least two colluding parties would need to run a malicious mcTLS implementation. Major browsers and Web servers (especially open source ones) are unlikely to do this, since they would almost surely be caught. Mobile apps using HTTP would need to implement HTTP themselves instead of using the platform's (honest) HTTP library. If it is essential to know that parties have not shared keys with unauthorized parties, some sort of remote attestation is the most promising solution (we describe remote attestation in Section 4.3.3).

Finally, even when all parties are honest, adding more entities to a session necessarily increases the attack surface: a bug or misconfiguration on any one could compromise the session. This risk is inherent in the problem, not any particular solution.

3.2.3 Design Overview

To satisfy the five design requirements, we add two key features to TLS:

(1) Encryption Contexts *(R2, R3, R5)* In TLS, there are only two parties, so it makes no sense to restrict one party's access to part of the data. With trusted middleboxes, however, the endpoints may wish to limit a middlebox's access to only a portion of the data or grant it read-only access. To make this possible, mcTLS introduces the notion of *encryption contexts*, or *contexts*, to TLS. An encryption context is simply a set of symmetric encryption and message authentication code (MAC) keys for controlling who can read and write the data sent in that context (Section 3.2.4). Applications can associate each context with a purpose (opaque to mcTLS itself) and access permissions for each middlebox. For instance, web browsers/servers could use one context for HTTP headers and another for content. We describe several strategies for using contexts in Section 3.3.2.

(2) Contributory Context Keys *(R1, R4)* The client and server each perform a key exchange with each middlebox after verifying the middleboxes' certificates if they choose to (R1). Next, the endpoints each generate half of every context key and send to each middlebox the half-keys for the contexts to which it has access, encrypted with the symmetric keys derived above. The middlebox only gains access to a context if it receives both halves of the key, ensuring that the client and server are both aware of each middlebox and agree on its access permissions (R4). The server may relinquish this control to avoid extra computation if it wishes (Section 3.2.6).

3.2.4 The mcTLS Record Protocol

The TLS record protocol takes data from higher layers (e.g., the application), breaks it into "manageable" blocks, optionally compresses, encrypts, and then MAC-protects each block, and finally transmits the blocks. mcTLS works much the same way, though each mcTLS record contains only

data associated with a single context; we add a one byte context ID to the TLS record format. Record sequence numbers are global across contexts to ensure the correct ordering of all application data at the client and server and to prevent an adversary from deleting an entire record undetected. Any of the standard encryption and MAC algorithms supported by TLS can be used to protect records in mcTLS. (So, details like the order of encryption and MAC depend on the cipher suite; mcTLS makes no changes here.)

Building on [172, 173], mcTLS manages access to each context by controlling which middleboxes are given which context keys. For each context, there are four relevant parties, listed in decreasing order of privilege: *endpoints* (client and server), *writers* (middleboxes with write access to the context), *readers* (middleboxes with read-only access to the context), and *third parties* (blanket term for middleboxes with no access to the context, attackers, and bit flips during transmission). Changes by writers are *legal modifications* and changes by readers and third parties are *illegal modifications*. mcTLS achieves the following three access control properties:

1. Endpoints can limit read access to a context to writers and readers only.
2. Endpoints can detect legal and illegal modifications.
3. Writers can detect illegal modifications.

Controlling Read Access Each context has its own encryption key (called $K_{readers}$, described below). Possession of this key constitutes read access, so mcTLS can prevent a middlebox from reading a context by withholding that context's key.

Controlling Write Access Write access is controlled by limiting who can generate a valid MAC. mcTLS takes the following "endpoint-writer-reader" approach to MACs. Each mcTLS record carries three keyed MACs, generated with keys $K_{endpoints}$ (shared by endpoints), $K_{writers}$ (shared by endpoints and writers), and $K_{readers}$ (shared by endpoints, writers, and readers). Each context has its own $K_{writers}$ and $K_{readers}$ but there is only one $K_{endpoints}$ for the session since the endpoints have access to all contexts.

Generating MACs

- When an **endpoint** assembles a record, it includes three MACs, one with each key.
- When a **writer** modifies a record, it generates new MACs with $K_{writers}$ and $K_{readers}$ and simply forwards the original $K_{endpoints}$ MAC.
- When a **reader** forwards a record, it leaves all three MACs unmodified.

Checking MACs

- When an **endpoint** receives a record, it checks the $K_{writers}$ MAC to confirm that no illegal modifications were made and it checks the $K_{endpoints}$ MAC to find out if any legal modifications were made (if the application cares).
- When a **writer** receives a record, it checks the $K_{writers}$ MAC to verify that no illegal modifications have been made.
- When a **reader** receives a record, it uses the $K_{readers}$ MAC to check if any *third party modifications* have been made.

Notation	Meaning
DH_E^+, DH_E^-	Entity E's ephemeral Diffie-Hellman public/private key pair
$DHCombine(\cdot,\cdot)$	Combine Diffie-Hellman public and private keys to produce a shared secret
PK_E^+, PK_E^-	Entity E's long-term signing public/private key pair (e.g., RSA)
$Sign_{PK_E^-}(\cdot)$	Signature using E's private key
S_E	Secret known only to entity E
$PS_{E_1\text{-}E_2}$	Pre-secret shared by entities E_1 & E_2
$S_{E_1\text{-}E_2}$	Secret shared between entities E_1 & E_2
$PRF_S(\cdot)$	Pseudorandom function keyed with secret S as defined in the TLS RFC [94]
$K_{E_1\text{-}E_2}$	Symmetric key shared by E_1 and E_2
K^E	Key material generated by entity E
$Enc_K(\cdot)$	Symmetric encryption using key K
$MAC_K(\cdot)$	Message authentication code with key K
$AuthEnc_K(\cdot)$	Authenticated encryption with key K
$H(\cdot)$	Cryptographic hash
$\|$	Concatenation

Table 3.2: **Notation.** Notation used in this chapter.

Note that with the endpoint-writer-reader MAC scheme, *readers cannot detect illegal changes made by other readers*. The problem is that a shared key cannot be used by an entity to police other entities at the same privilege level. Because all readers share $K_{readers}$ (so that they can detect third party modifications), all readers are also capable of generating valid $K_{readers}$ MACs. This is only an issue when there are two or more readers for a context and, in general, readers not detecting reader modifications should not be a problem (reader modifications are still detectable at the next writer or endpoint). However, if needed, there are two options for fixing this: (a) readers and writers/endpoints share pairwise symmetric keys; writers/endpoints compute and append a MAC for *each* reader, or (b) endpoints and writers append digital signatures rather than MACs; unlike $K_{writers}$ MACs, readers can verify these signatures. The benefits seem insufficient to justify the additional overhead of (a) or (b), but they could be implemented as optional modes negotiated during the handshake.

3.2.5 The mcTLS Handshake Protocol

The mcTLS handshake is very similar to the TLS handshake. We make two simplifications here for ease of exposition: first, although the principles of the mcTLS handshake apply to many of the cipher suites available in TLS, we describe the handshake using ephemeral Diffie-Hellman with RSA signing keys because it is straightforward to illustrate and common in practice. Second, we describe the handshake with a single middlebox, but extending it to multiple middleboxes is straightforward. Table 3.2 defines the notation we use in this chapter.

The purpose of the handshake is to:

- Allow the endpoints to agree on a cipher suite, a set of encryption contexts, a list of middle-

	mcTLS			mcTLS (Client Key Dist.)			Split TLS		
	Client	*Middlebox*	*Server*	*Client*	*Middlebox*	*Server*	*Client*	*Middlebox*	*Server*
Hash	$12 + 6N$	0	$12 + 6N$	$10 + 5N$	0	$10 + 5N$	10	20	10
Secret Comp.	$N + 1$	2	$N + 1$	$N + 1$	1	1	1	2	1
Key Gen.	$4K + N + 1$	$(k \leq 2K) + 2$	$4K + N + 1$	$2K + N + 1$	1	1	1	2	1
Asym Verify	$N + 1$	$n \leq 1$	$n \leq N$	$N + 1$	$n \leq 1$	0	1	1	0
Sym Encrypt	$N + 2$	0	$N + 2$	$N + 2$	0	1	1	2	1
Sym Decrypt	2	2	2	1	1	2	1	2	1

(N = number of middleboxes; K = number of contexts)

Table 3.3: **mcTLS Handshake Crypto.** Cryptographic operations performed by the client, middlebox, and server during the handshake. Assumes no TLS extensions, a DHE-RSA cipher suite, and the client is not authenticated with a certificate.

boxes, and permissions for those middleboxes.

- Allow the endpoints to authenticate each other and all of the middleboxes (if they choose to).
- Establish a shared symmetric key $K_{endpoints}$ between the endpoints.
- Establish a shared symmetric key $K_{writers}$ for each context among all writers and a shared symmetric key $K_{readers}$[1] for each context among all readers.

Handshake. Below we explain the steps of a mcTLS handshake (shown in Figure 3.1), highlighting the differences from TLS. Note that it has the same 2-RTT "shape" as TLS.

1. **Setup:** Each party generates a (public) random value and an ephemeral Diffie-Hellman key pair (the middlebox generates two key pairs, one for the client and one for the server). The endpoints also each generate a secret value.
2. **Client Hello:** Like TLS, an mcTLS session begins with a `ClientHello` message containing a random value. In mcTLS, the `ClientHello` carries a `MiddleboxListExtension`, which contains (1) a list of the middleboxes to include in the session—we discuss building this list in the first place in Section 3.5.1—and (2) a list of encryption contexts, their purposes (strings meaningful to the application), and middlebox access permissions for every context. The client opens a TCP connection with the middlebox and sends the `ClientHello`; the middlebox opens a TCP connection with the server and forwards the `ClientHello`.
3. **Certificate & Public Key Exchange:** As in TLS, the server responds with a series of messages containing a random value, its certificate, and an ephemeral public key signed by the key in the certificate. The middlebox does the same: it sends its random value, certificate, and ephemeral public key to both the client and the server. The client sends an ephemeral public key, which the middlebox saves and forwards to the server. The middlebox piggy-backs its messages on the `ServerKeyExchange` and `ClientKeyExchange` messages (indicated by dashed arrows). The ephemeral keys provide forward secrecy; the middlebox uses different key pairs for the client and the server to prevent small subgroup attacks [166].
4. **Shared Key Computation:** The **client** computes two secrets ($S_{C\text{-}M}$ and $S_{C\text{-}S}$) using the

[1]Though we describe each as one key for simplicity, $K_{readers}$ and $K_{endpoints}$ are really four keys each (just like the "session key" in TLS): an encryption key for data in each direction and a MAC key for data in each direction. Likewise, $K_{writers}$ is really one MAC key for each direction.

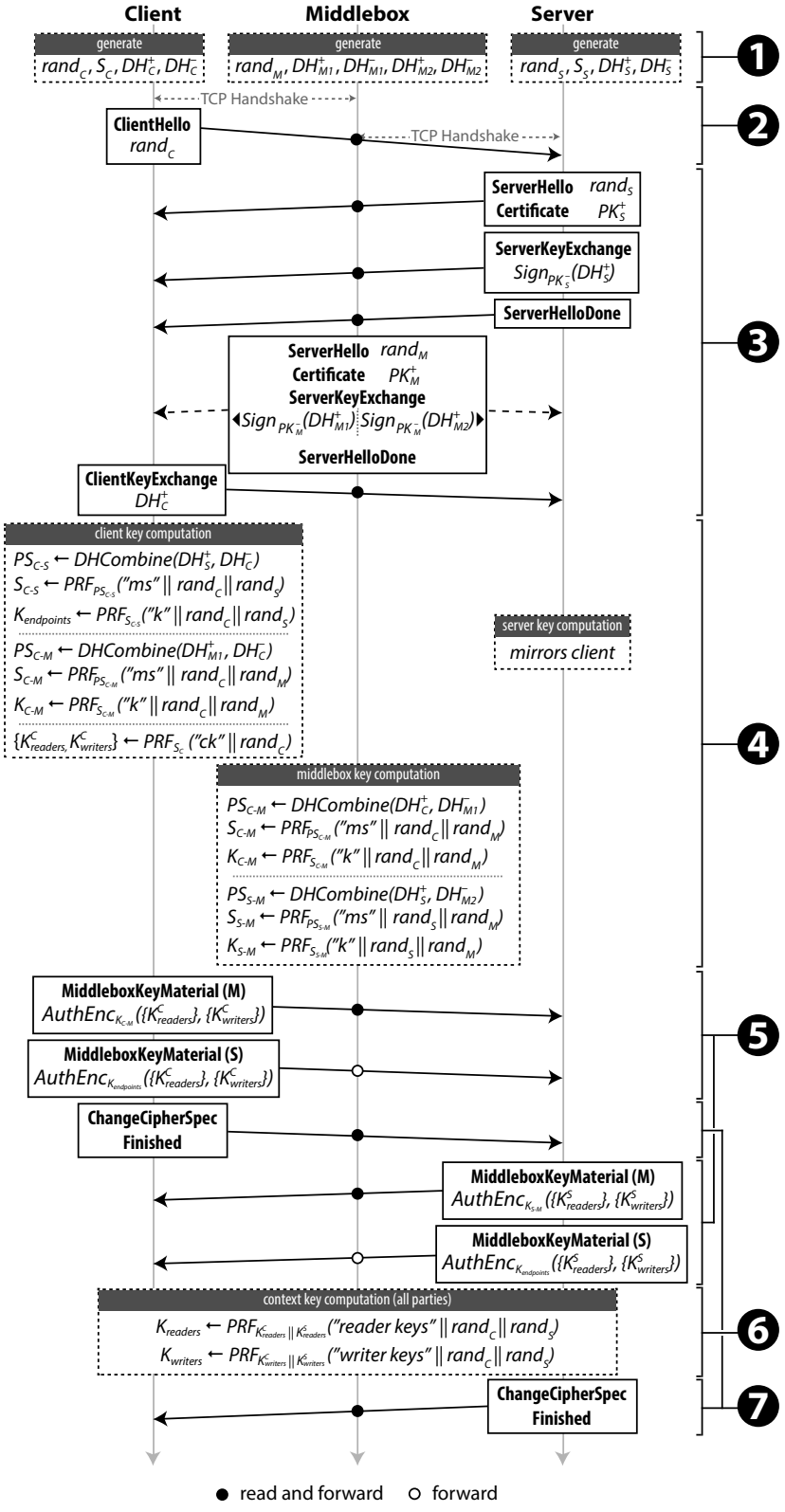


Figure 3.1: **mcTLS Handshake.**

contributions from the server and middlebox, which it uses to generate a symmetric key shared with the middlebox ($K_{C\text{-}M}$) and the server ($K_{endpoints}$). The client also generates "partial keys," $K^C_{writers}$ and $K^C_{readers}$, for each context, using a secret known only to itself.
The **server** follows the same procedure as the client, resulting in $K_{endpoints}$, $K_{S\text{-}M}$, $K^S_{writers}$, and $K^S_{readers}$. The server may choose to avoid this overhead by asking the client to generate and distribute complete context keys (Section 3.2.6).
When the **middlebox** receives the `ClientKeyExchange`, it computes $K_{C\text{-}M}$ and $K_{S\text{-}M}$ using the client's and server's ephemeral public keys, respectively; it will use these keys later to decrypt context key material from the client and server.

5. **Context Key Exchange:** Next, for each context, the endpoints send the partial context keys to the middlebox ($K^C_{readers}$ and $K^S_{readers}$ if it has read access and $K^C_{writers}$ and $K^S_{writers}$ if it has write access). These messages are sent encrypted and authenticated under $K_{C\text{-}M}$ and $K_{S\text{-}M}$, ensuring the secrecy and integrity of the partial context keys. The middlebox forwards each message on to the opposite endpoint so it can be included in the hash of the handshake messages that is verified at the end of the handshake. The endpoints also send all of the partial context keys to the opposite endpoint encrypted under $K_{endpoints}$. The middlebox forwards this message (but cannot read it).
6. **Context Key Computation:** The client indicates that the cipher negotiated in the handshake should be used by sending a `ChangeCipherSpec` message. Receipt of the `ChangeCipherSpec` message prompts all parties to generate context keys using $PRF(\cdot)$ keyed with the concatenation of the partial context keys. This "partial key" approach serves two purposes: (1) it provides contributory key agreement (both endpoints contribute randomness) and (2) it ensures that a middlebox only gains access to a context if the client and server both agree.
7. **Finished:** The mcTLS handshake concludes with the exchange of `Finished` messages. As in TLS, the `Finished` message contains a hash of the concatenation of all handshake messages (including those directed to the middlebox): $PRF_{S_{C\text{-}S}}(finished\_label, H(messages))$. Verifying this message ensures that both endpoints observe the same sequence of identical handshake messages, i.e., no messages were modified in flight.

Details. There are a few subtle differences between the mcTLS and TLS handshakes. We briefly highlight the changes here and argue why they are safe; for a more detailed security analysis, see [1].

- For simplicity, the middlebox cannot negotiate session parameters (e.g., cipher suite or number of contexts). A more complex negotiation protocol could be considered in future work if needed.
- The server's context key material is not included in the client's `Finished` message, since this would require an extra RTT. However, this key material is sent encrypted and MAC-protected, so an adversary cannot learn or modify it.
- The client cannot decrypt the context key material the server sends the middlebox and vice versa. This would require establishing a three-way symmetric key between both endpoints and each middlebox. Because the middlebox needs key material from *both* endpoints, one rogue endpoint cannot unilaterally increase a middlebox's permissions.

- The middlebox cannot verify the handshake hash in the `Finished` message because it does not know $K_{endpoints}$. We do not include per-middlebox `Finished` messages to avoid overhead. This means it is possible for the middlebox to observe an incorrect sequence of handshake messages. However, this is at most a denial of service attack. For instance, even though the middlebox cannot detect a cipher suite downgrade attack, the endpoints would detect it and terminate the session. Furthermore, context key material is sent encrypted and MAC-protected under keys each endpoint shares with the middlebox, so as long as at least one endpoint verifies the middlebox's certificate, an adversary cannot learn or modify the context keys.

3.2.6 Reducing Server Overhead

One concern (albeit a diminishing one [51, 152]) about deploying TLS is that the handshake is computationally demanding, limiting the number of new connections per second servers can process. We do not want to make this problem worse in mcTLS, and one way we avoid this is by making certain features optional. For example, similar to TLS, authentication of the entities in the session is optional—in some cases, the server may not care who the middleboxes are. Another burden for servers in mcTLS is generating and encrypting the partial context keys for distribution to middleboxes. Rather than splitting this work between the client and server, it can optionally be moved to the client: context keys are generated from the master secret shared by the endpoints and the client encrypts and distributes them to middleboxes ("client key distribution mode"). This reduces the server load, but it has the disadvantage that agreement about middlebox permissions is not enforced. (Note that this does not sacrifice contributory key agreement in the sense that both endpoints contribute randomness. The client generates the context keys from the secret it shares with the server; if client/server key exchange was contributory, the context keys inherit this benefit.) Choosing a handshake mode is left to content providers, who can individually decide how to make this control-performance tradeoff; servers indicate their choice to clients in the `ServerHello`.

Table 3.3 compares the number of cryptographic operations performed by mcTLS and the split TLS approach described in Section II.3.2. We show numbers for mcTLS both without and with client context key distribution. If we consider a simple example with a single middlebox ($N = 1$), the additional server load using client key distribution mode is limited to a small number of lightweight operations (Hash and Sym Decrypt).

3.3 Using mcTLS

3.3.1 Using Contexts

Just as the architects of HTTP had to define how it would operate over TLS [195], protocol designers need to standardize how their applications will use mcTLS. From an application developer's perspective, the biggest change mcTLS brings is contexts: the application needs to decide how many contexts to use and for what. First we give the topic a general treatment and then follow up with some concrete use cases below.

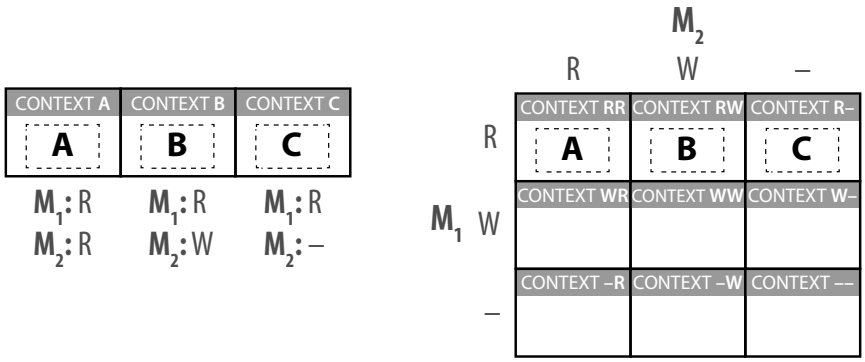


Figure 3.2: **Controlling Permissions with Encryption Contexts.** Strategies for using encryption contexts: context-per-section (left) and context-as-permissions (right).

There are two ways to think about contexts: as sections of the data stream or as configurations of middlebox permissions. For example, suppose an application wants to transfer a document consisting of three pieces, A, B, and C, via two middleboxes, M_1 and M_2. M_1 should have read access to the entire document and M_2 should read A, write B, and have no access to C. The application could allocate one context for each piece and assign the appropriate permissions (Figure 3.2 left), or it could create one context for each combination of permissions and use the appropriate context when sending each piece of the document (Figure 3.2 right).

Which model is appropriate depends on the use case: in the context-per-section model, n sections means n contexts. In the contexts-as-permissions model, m middleboxes means 3^m contexts. In practice, we expect at least one of these numbers to be small, since data in a session often is not of wildly varying levels of sensitivity and since most middleboxes need similar permissions (Table 3.1). For instance, in the case of HTTP, we imagine four contexts will be sufficient: request headers, request body, response headers, and response body. (Though you could imagine extreme cases in which each HTTP header has its own access control settings.)

Finally, the example above uses a static context assignment, but contexts can also be selected dynamically. An application could make two contexts, one which a middlebox can read and one it cannot, and switch between them to enable or disable middlebox access on-the-fly (for instance, to enable compression in response to particular user-agents).

3.3.2 Use Cases

Data Compression Proxy. Many users—particularly on mobile devices—use proxies like Chrome's Data Compression Proxy [37], which re-scale/re-encode images, to reduce their data usage. However, Google's proxy currently ignores HTTPS flows. With mcTLS, users can instruct their browsers to give the compression proxy write access to HTTP responses. One step further, the browser and web server could coordinate to use two contexts for responses: one for images, which the proxy can access, and the other for HTML, CSS, and scripts, which the proxy cannot access. Context assignments can even change dynamically: if a mobile user connects to Wi-Fi mid-page-load, images might also be transferred over the no-access context since compression is no longer required.

Parental Filtering. Libraries and schools—and sometimes even entire countries [70]—often employ filters to block age-inappropriate content. Such filters often depend on seeing the full URL being accessed (only 5% of the entries on the Internet Watch Foundation's blacklist are entire (sub-)domains [175]). With mcTLS, IT staff could configure their machines to allow their filter read-only access to HTTP request headers, and user-owned devices connecting to the network could be configured to do the same dynamically via DHCP. The filter drops non-compliant connections.

Corporate Firewall. Most companies funnel all network traffic through intrusion detection systems (IDS)/firewalls/virus scanners. Currently, these devices either ignore encrypted traffic or install root certificates on employees' devices, transparently giving themselves access to all "secure" sessions. With mcTLS, administrators can configure devices to give the IDS—which users can now see—read-only access. Security appliances no longer need to impersonate end servers and users no longer grow accustomed to installing root certificates.

Online Banking. Though we designed mcTLS to give users control over their sessions, there are cases in which the content provider really does know better than the user and should be able to say "no" to middleboxes. A prime example is online banking: banks have a responsibility to protect careless or nontechnical users from sharing their financial information with third parties. The server can easily prevent this by simply not giving middleboxes its half of the context keys, regardless of what level of access the client assigns.

HTTP/2 Streams. One of the features of HTTP/2 is multiplexing multiple streams over a single transport connection. mcTLS allows browsers to easily set different access controls for each stream.

3.4 Evaluation

mcTLS' fine-grained access control requires generating and distributing extra keys, computing extra MACs, and, possibly, sending a larger number of smaller records than TLS. In this section we evaluate this overhead.

Experimental Setup. We built a prototype of mcTLS by modifying the OpenSSL[2] implementation of TLSv1.2. The prototype supports all the features of mcTLS's default mode, described in Section 3.2.4 and Section 3.2.5. We use the DHE-RSA-AES128-SHA256 cipher suite, though nothing prevents mcTLS from working with any standard key exchange, encryption, or MAC algorithm. In addition, though the `MiddleboxKeyMaterial` message should be encrypted using a key generated from the DHE key exchange between the endpoints and the middlebox, we use RSA public key cryptography for simplicity in our implementation. As a result, *forward secrecy* is not currently supported by our implementation. mcTLS requires some additions to the API, e.g., defining contexts and their read/write permissions, but these are similar to the current OpenSSL API.

[2] OpenSSL v1.0.1j from October 2014

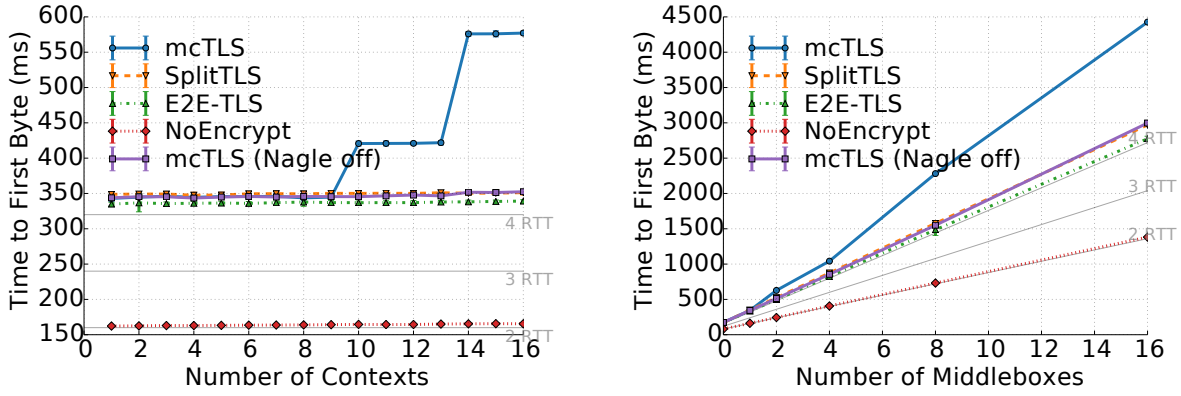


Figure 3.3: Time to first byte vs. # contexts (left) and # middleboxes (right).

Next we wrote a simple HTTP client, server, and proxy that support four modes of operation:

1. `mcTLS`: Data transferred using mcTLS.
2. `SplitTLS`: Split TLS connections between hops; middleboxes decrypt and re-encrypt data.
3. `E2E-TLS`: A single end-to-end TLS connection; middleboxes blindly forward encrypted data.
4. `NoEncrypt`: No encryption; data transferred and forwarded in the clear over TCP.

We instrumented the mcTLS library and our applications to measure handshake duration, file transfer time, data volume overhead, and connections per second.

We test in two environments. (1) *Controlled*: Client, middleboxes, and server all run on a single machine. We control bandwidth (10 Mbps unless otherwise noted, chosen from the median of SpeedTest.net samples) and latency with `tc`. (2) *Wide Area*: We run the client, middlebox, and server on EC2 instances in Spain, Ireland, and California, respectively. The client connects either over fiber or 3G. Unless otherwise specified, experiments in either environment consist of 50 runs for which we report the mean; error bars indicate one standard deviation. Middleboxes are given full read/write access to each context since this is the worst case for mcTLS performance.

3.4.1 Time Overhead

Handshake Time. Figure 3.3 (left) shows the time to first byte as the number of contexts increases. There is one middlebox and each link has a 20 ms delay (80 ms total RTT). `NoEncrypt` serves as a baseline, with a time to first byte of 160 ms, or 2 RTT. Up to 9 contexts, `mcTLS`, `E2E-TLS`, and `SplitTLS` each take 4 RTTs. At 10 contexts, `mcTLS` jumps to 5 RTT and at 14 to 7.

The culprit was TCP's Nagle algorithm, which delays the transmission of data until a full MSS is ready to be sent. At 10 contexts, the handshake messages from the proxy to the server exceed 1 MSS and Nagle holds the extra bytes until the first MSS is ACKed. At 14 contexts the same thing happens to the middlebox key material from the client (+1 RTT) and the server (+1 RTT). Disabling the Nagle algorithm (not uncommon in practice [170]) solved the problem. We tried `E2E-TLS`, `SplitTLS`, and `NoEncrypt` without Nagle as well, but their performance did not improve since their messages never exceed 1 MSS.

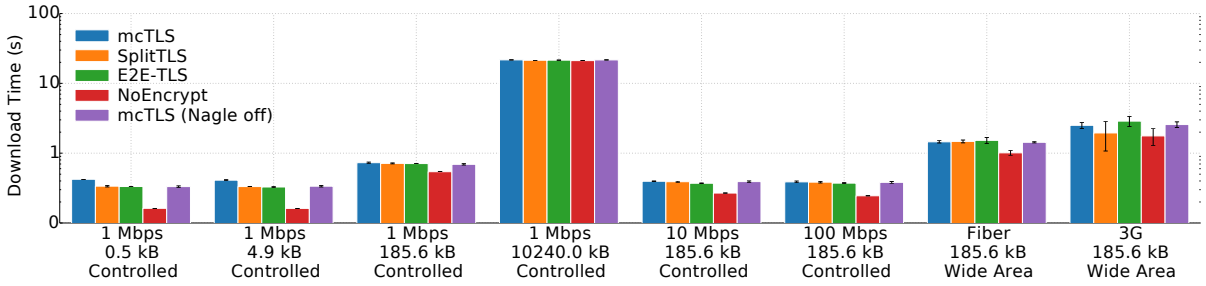


Figure 3.4: **Transfer Time.** File download time for various configurations of link speed and file size.

Time to first byte scales linearly with the number of middleboxes, since in our experiments adding a middlebox also adds a 20 ms link (Figure 3.3 right). The latency increase and the extra key material to distribute exacerbate the problems caused by Nagle; disabling it once again brings `mcTLS` performance in line with `E2E-TLS` and `SplitTLS`. Finally, if middleboxes lie directly on the data path (which often happens), then the only additional overhead is processing time, which is minimal.

Takeaway: *mcTLS's handshake is not discernibly longer than `SplitTLS`'s or `E2E-TLS`'s.*

File Transfer Time. Next we explore the timing behavior of each full protocol by transferring files through a single middlebox. To choose realistic file sizes, we loaded the top 500 Alexa pages and picked the 10th, 50th, and 99th percentile object sizes (0.5 kB, 4.9 kB, and 185 kB, respectively). We also consider large (10MB) downloads (e.g., larger zip files or video chunks).

The first four bar groups in Figure 3.4 show the download time for increasing file sizes at 1 Mbps; each bar represents 10 repetitions. As expected, the handshake overhead dominates for smaller files (<5 kB); all protocols that use encryption require an additional ~17 ms compared to `NoEncrypt`. mcTLS is comparable to `E2E-TLS` and `SplitTLS`. We see the same behavior when downloading files at different link rates or in the wide area (last four bar groups). Handshake and data transfer dominate download time; protocol-specific processing makes little difference.

Takeaway: *`mcTLS` transfer times are not substantially higher than `SplitTLS` or `E2E-TLS` irrespective of link type, bandwidth, or file size.*

Page Load Time. To understand how the micro-benchmarks above translate to real-world performance, we examine web page load time. Though we have not yet ported a full-blown web browser to mcTLS, we approximate a full page load in our simple client as follows. First, we load all of the Alexa top 500 pages that support HTTPS in Chrome. For each page, we extract a list of the objects loaded, their sizes, and whether or not an existing connection was re-used to fetch each one (we cannot tell *which* connection was used, so we assign the object to an existing one chosen at random). Next, our client "plays back" the page load by requesting dummy objects of the appropriate sizes from the server. We make the simplifying assumption that each object depends only on the previous object loaded in the same connection (this might introduce false dependencies and ignore true ones).

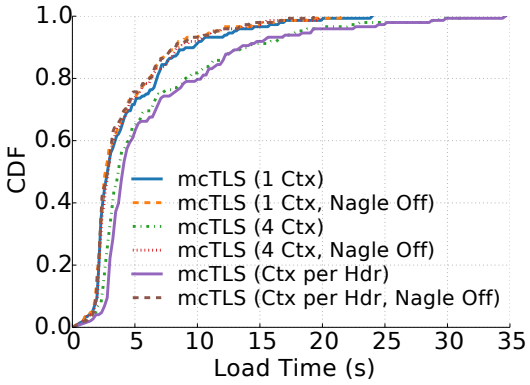


Figure 3.5: Page load time for different numbers of mcTLS contexts.

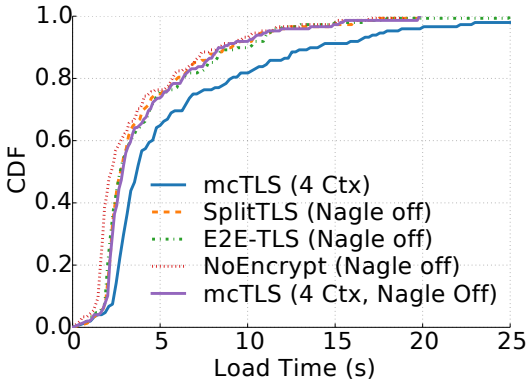


Figure 3.6: Page load time for different protocols.

First, we compare three mcTLS strategies: 1-Context (all data in one context), 4-Context (request headers, request body, response headers, response body), and ContextPerHeader (one context for each HTTP header, one for request body, and one for response body). Figure 3.5 shows the CDF of page load times for each strategy. The plot shows similar performance for each strategy, indicating that mcTLS is not overly sensitive to the way data is placed into contexts.

Next we compare `mcTLS` to `SplitTLS`, `E2E-TLS`, and `NoEncrypt` (Figure 3.6). We use the 4-Context strategy for `mcTLS`, since we imagine it will be the most common. `SplitTLS`, `E2E-TLS`, and `NoEncrypt` perform the same, while mcTLS adds a half second or more. Once again, Nagle is to blame: sending data in multiple contexts causes back-to-back `send()` calls to TCP. The first record is sent immediately, but the subsequent records are held because they are smaller than an MSS and there is unacknowledged data in flight. Repeating the experiment with Nagle turned off closed the gap.

Takeaway: *mcTLS has* no *impact on real world Web page load times.*

3.4.2 Data Volume Overhead

Most of mcTLS' data overhead comes from the handshake, and it increases with the number of middleboxes and contexts (due to certificates and context key distribution). Figure 3.7 shows the total size of the handshake for different numbers of contexts and middleboxes. For a base configuration with one context and no middleboxes, the `mcTLS` handshake is 2.1 kB compared to 1.6 kB for `SplitTLS` and `E2E-TLS`. Note that the handshake size is independent of the file size.

Next, each record carries MACs (three in mcTLS, one in TLS). Their impact depends on the application's sending pattern—smaller records mean larger overhead. For the web browsing experiments in Section 3.4.1, the median MAC overhead for `SplitTLS` compared to `NoEncrypt` was 0.6%; as expected, `mcTLS` triples that to 2.4%.

Finally, padding and header overhead are negligible.

Takeaway: *Apart from the initial handshake overhead, which is negligible for all but short connections, mcTLS introduces less than 2% additional overhead for web browsing compared to `SplitTLS` or `E2E-TLS`.*

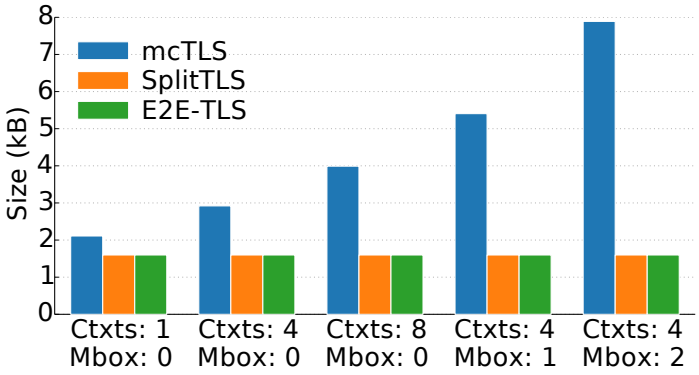


Figure 3.7: **Handshake Sizes.** mcTLS handshake size for varying numbers of encryption contexts and middleboxes.

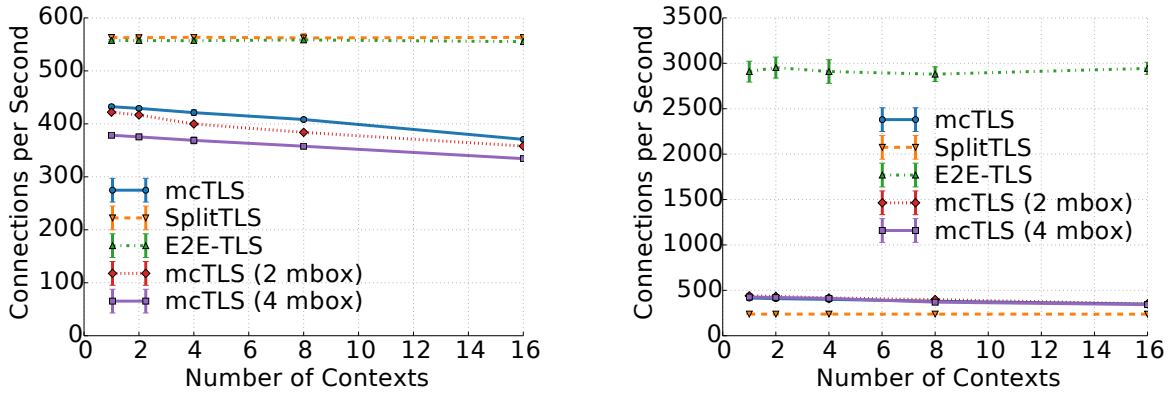


Figure 3.8: Load sustainable at the server (left) and middlebox (right).

3.4.3 CPU Overhead

Figure 3.8 (left) shows the number of connections (only handshakes) per second the server can sustain. We see that the extra asymmetric encryption for distributing middlebox key material takes a toll. The `mcTLS` server handles 23% fewer connections than `SplitTLS` or `E2E-TLS`; that number drops to 35% fewer as the number of contexts, and therefore the number of partial context keys the server must encrypt, increases. We note two things: (1) key distribution optimizations, which we intend to pursue in future work, can shrink this gap, and (2) the server can reclaim this lost performance if the client handles key generation/distribution (Section 3.2.6).

The results for the middlebox are more interesting (Figure 3.8 right). First, `E2E-TLS` significantly outperforms `mcTLS` and `SplitTLS` (note the change in Y scale) because it does not participate in a TLS handshake. Second, `mcTLS` performs *better* than `SplitTLS` because in `SplitTLS` the proxy has to participate in *two* TLS handshakes. These results show it is not only feasible, but practical to use middleboxes in the core network.

Takeaway: *`mcTLS` servers can serve 23%–35% fewer connections per second than `SplitTLS`, but `mcTLS` middleboxes can serve 45%–75% more.*

3.4.4 Deployment

To begin understanding deployability, we built an extension to the Ruby SSL library that adds support for mcTLS with less than 250 lines of C code. Using the extension, we then built a 17 line Ruby web client with the same functionality as our C/OpenSSL-based evaluation client. While a bit more work is needed to make the extension more Ruby-like, the potential to easily write mcTLS-enabled mobile apps with developer-friendly tools like RubyMotion [2] is promising.

We also modified the OpenSSL `s_time` benchmarking tool to support mcTLS. Again, minimal changes were required: less than 30 new lines of C code were added, and about 10 lines were slightly changed. This means that relatively minor developer effort is required to gain the full benefits of mcTLS.

While supporting fine-grained access control requires the minimal effort of assigning data to a context and setting middlebox permissions for those contexts, many of the benefits of mcTLS are immediately available with just support from the HTTP client library and server. For example, HTTP libraries could use the 4-Context strategy by default, requiring no additional programming or effort from application developers. Finally, we note that clients and servers can easily fall back to regular TLS if an mcTLS connection cannot be negotiated.

Takeaway: *Upgrading an application or library to mcTLS appears to be straightforward and easy.*

3.5 Discussion

3.5.1 Middlebox Discovery

mcTLS assumes that the client has a list of middleboxes prior to initiating a handshake, which it includes in the `ClientHello`. Building this list is largely orthogonal to mcTLS itself; many existing mechanisms could be used, depending on *who* is trying to add a middlebox to the session. For example:

- **Users** or **system administrators** might configure the client (application or OS) directly (e.g., the user might point his browser toward Google's SPDY proxy). If users express interest in, e.g., a "nearby" data compression proxy, rather than a particular one, clients could discover available proxies using mDNS [79] or DNS-SD [78].
- **Content providers** could specify middleboxes to be used in any connection to its servers using DNS.
- **Network operators** can use DHCP or PDP/PDN to inform clients of any required middleboxes (e.g., virus scanners).

If a priori mechanisms like these are not flexible enough, the handshake could be extended to allow, e.g., on-path middleboxes to insert themselves (subject to subsequent approval by the endpoints, of course) during session setup. The costs and benefits of this are not immediately clear; we leave working out the details of more complex session negotiation for future work.

	R1	R2	R3	R4	R5
mcTLS	•	•	•	•	•
(1) Custom Certificate					
(2) Proxy Certificate Flag	○			○	
(3) Session Key Out-of-Band	•	•		○	
(4) Custom Browser					
(5) Proxy Server Extension	○	○	○	○	

(• = full compliance; ○ = partial compliance)

Table 3.4: **Comparison of Solutions.** Design principle compliance for mcTLS and competing proposals.

3.5.2 User Interface

The technical solution for adding middleboxes to secure communication sessions means little without suitable interfaces through which users can control it. The primary challenges for such an interface are:

- *Indicating to the user that the session is "secure."* Re-using the well-known lock icon is misleading, since the semantics of TLS and mcTLS differ.
- *Communicating to the user who can do what.* Which middleboxes can read the user's data? Which can modify it? What modifications do they make? Who owns the middleboxes? Who added them to the session and why?
- *Allowing users to set access controls.* Which sessions can a middlebox see? Within those sessions, which fields can it read or write? The difficulty is making such controls simple and scalable. For instance, asking users to set middlebox permissions for each domain they visit is not practical.

Designing a satisfactory interface atop mcTLS is a project in and of itself, one we cannot begin to do justice here.

3.6 Comparison to Related Work

There has been a lot of recent interest, particularly in industry, for including intermediaries in encrypted sessions. Section II.3 describes several such proposals. Below we compare them to mcTLS in terms of our five design requirements (summarized in Table 3.4).

(1) Custom Root Certificate. This technique does not meet any of our requirements. First, the server, and in many cases the client, is not aware of the existence of the middlebox (R4) so it clearly cannot authenticate it (R1). Second, the middlebox has full read and write access to all data in the session (R5). Finally, since the client has no control after the first hop, there is no guarantee about the secrecy, integrity, or authenticity of the data (R2, R3) or the identity of the server (R1).

(2) "I'm a proxy" Certificate Flag. In this case, the client is made explicitly aware of the presence of the middlebox, so it can authenticate it (R1) and can control its use on a per connection basis

(R4). The client still cannot authenticate the server and the server is unaware of the middlebox. R2, R3, and R5 remain unaddressed.

(3) Pass Session Key Out-of-Band. Compared with (1), this solution has the additional benefit that the client authenticates both the middlebox and the server (R1) and knows that the session is encrypted end-to-end (R2). R3, R4, and R5 are still partially or completely unaddressed.

(4) Ship a Custom Browser. This solution is essentially the same as (1), so it also fails all requirements. In addition, it has the drawback that a custom browser may not be updated quickly, is expensive to develop and maintain, and may be inconvenient to users.

(5) Proxy Server Extension. The client must *completely* trust the middlebox to provide honest information about the server certificate and ciphersuite, so this solution only partially fulfills R1, R2, and R3. The proxy is not necessarily visible to the server, so only partial R4. Finally, the proxy has read/write access to all data (R5).

Other Approaches. An alternative to TLS-based techniques is an extension to IPsec that allows portions of the payload to be encrypted/authenticated between the two end-points of a security association and leaves the remainder in the clear [144]. This solution leaves data for middleboxes completely unencrypted (R2); R1 and R3 are also violated. Furthermore, this approach does not allow explicit control of the data flow to different entities (R4).

Tcpcrypt [62, 63] is an alternative proposal for establishing end-to-end encrypted sessions. Similar to TLS, tcpcrypt supports communication between two endpoints only, but we believe that the concepts of encryption contexts and contributory context keys could be applied to it as well. However, because of mcTLS's increased handshake size, it may no longer be possible to embed the entire handshake in the TCP handshake.

3.7 Conclusion

The increasing use of TLS provides privacy and security, but also leads to the loss of capabilities that are typically provided by an invisible army of middleboxes offering security, compression, caching, or content/network resource optimization. Finding an incrementally deployable solution that can bring back these benefits while maintaining the security expectations of clients, content providers, and network operators is not easy. mcTLS does this by extending TLS, which already carries a significant portion of HTTP traffic. mcTLS focuses on transparency and control: (1) trusted middleboxes are introduced at the consent of both client and server, (2) on a per session basis, (3) with clear access rights (read/write), and (4) to specific parts of the data stream.

We show that building such a protocol is not only feasible but also introduces limited overhead in terms of latency, load time, and data overhead. More importantly, mcTLS can be incrementally deployed and requires only minor modifications to client and server software to support the majority of expected use cases. By using mcTLS, secure communication sessions can regain lost efficiencies with explicit consent from users and content providers.

Chapter 4 Outsourced Middleboxes and Legacy Clients in TLS

In Chapter 3, we presented Multi-Context TLS, which introduced access control for middleboxes. Unfortunately, in doing so, mcTLS requires that *both* endpoints be upgraded, which makes it unlikely to be deployed in the immediate future. This raises the question: what properties are most important for secure communication sessions with middleboxes? Is access control crucial? Is it worth the slow rollout? Are there easier-to-deploy protocols that still provide "enough" security? Despite initial efforts in both industry and academia, we remain unsure how to integrate middleboxes into secure sessions—it is not even clear how to *define* "secure" in this multi-party context.

So, in this chapter, we first step back and describe a design space for secure communication protocols for more than two parties, highlighting tradeoffs between mutually incompatible properties. We then target real-world requirements unmet by existing protocols, like *outsourcing middleboxes to untrusted infrastructure* and *supporting legacy clients*. We propose a security definition and present *Middlebox TLS (mbTLS)*, a protocol that provides it (in part by using Intel SGX to protect middleboxes from untrusted hardware). Finally, we show that mbTLS is deployable today and introduces low overhead, and we describe our experience building a simple mbTLS HTTP proxy.

4.1 Introduction

So far we have seen a number of solutions using middleboxes along with encryption, most notably the "split TLS" approach in widespread use today. We have also seen how approaches like split TLS drastically weaken overall security. These weaknesses underscore the fact that, while the properties of TLS are well-understood in the two-party case, it is unclear how to define "secure" in the multi-party case. In response, recent work has proposed new protocols alongside new security definitions. For example, Multi-Context TLS (mcTLS) (Chapter 3) allows endpoints to restrict which parts of the data stream the middleboxes can read or write and BlindBox [209] allows middleboxes to operate directly on encrypted data. However, these are largely "point solutions" that, while useful in certain scenarios, leave several real-world needs unmet.

In this chapter, we focus on three practical requirements that are so far unaddressed. First, there is increasing interest in outsourcing middlebox functionality to third-party cloud providers [6, 31, 151, 208] or to ISPs [14, 24, 33]. This setting poses a new challenge: *the owner of middlebox software ("middlebox service provider") and the owner of the hardware it runs on ("infrastructure provider")*

are not the same. If the infrastructure is untrusted, existing protocols like "split TLS" and mcTLS cannot provide the standard security properties TLS gives us today because (1) session data and keys are visible in memory and (2) the endpoints cannot tell if infrastructure provider actually ran the intended code. Second, in order to be realistically deployable, any new protocol should be reverse compatible with TLS. That is, if one endpoint wants to include a middlebox, it must be able to do so even while *inter-operating with legacy TLS endpoints*. And third, though less flashy, the ability to discover middleboxes on-the-fly is a practical requirement in many contexts. For example, if a service provider places proxies in edge ISPs, directing each client to connect to its local proxy using DNS (1) is an unnecessary configuration burden and (2) is brittle, since clients can use non-local DNS resolvers like OpenDNS.

We make two primary contributions in this chapter. First, we carefully articulate a **design space for secure multi-entity communication protocols** (and use it to place previous work in context). We describe the different properties that such a protocol might have and argue why some combinations are impossible to achieve at once, suggesting that the community needs to either carefully select which set of properties to support or develop different protocols for different use cases. Second, we present ***Middlebox TLS (mbTLS)***, a protocol for secure multi-entity communication that addresses the needs described above:

(1) mbTLS protects session data from third party infrastructure providers. mbTLS leverages trusted computing technology, like Intel SGX [39, 129, 165], to isolate the middlebox execution environment from the third party infrastructure.

(2) mbTLS interoperates with legacy TLS endpoints. Unlike mcTLS or BlindBox, an mbTLS endpoint can securely include middleboxes in a session with an unmodified TLS endpoint. In our tests, we successfully loaded content from more than 300 of the top Alexa sites using an mbTLS client.

(3) mbTLS provides other useful properties unique to multi-party settings. For example, mbTLS guarantees that data visits middleboxes in the order specified by the endpoints, prevents attackers from learning whether or not a middlebox modified a piece of data before forwarding it on, and provides in-band middlebox discovery.

We implement mbTLS using OpenSSL and the Intel SGX SDK and evaluate its performance, showing that (1) mbTLS adds no handshake latency compared to TLS, (2) mbTLS reduces CPU load on the middlebox and adds reasonable overhead on the server, and (3) running inside an SGX enclave does not degrade throughput.

Our hope is that mbTLS represents a significant and practical step toward bridging the gap between end-to-end security and the reality that middleboxes are not going away.

4.2 Multi-Entity Communication

Most network communication sessions today involve more parties than just a client and a server. By and large, these additional parties fall into one of three categories:

Network-Layer Middleboxes *(e.g., firewall, NAT, layer 3 load balancer).* They process data packet by packet and do not need to reconstruct or access application layer data.

Application-Layer Middleboxes *(e.g., virus scanner, IDS, parental filter, cache, compression proxy, application layer load balancer).* These *do* need access to application layer data.

Application-Layer Delegates *(e.g., CDNs).* In contrast to middleboxes, which act as intermediaries between client and server *at communication time*, we use the term *delegate* for intermediaries that take on the role of the server during the session (though in terms of real-world relationships, they are still more naturally viewed as intermediaries). Content delivery networks (CDNs) are a good example; clients talk to CDN servers and not directly with the origin server.

As our security practices improve and we move toward an Internet where encryption is ubiquitous, it is becoming clear that we do not have an adequate protocol for secure multi-entity communication, nor do we know exactly what properties one should provide. In the two-party case, it is well understood what security properties we want and how to achieve them; we have been using TLS successfully for years. But in the multi-party case, there are still two key unanswered questions: *(1) what security properties should hold for sessions involving three or more parties?* and *(2) what are the best mechanisms to enforce those properties?*

The answers to these questions will be different for each of the three categories of intermediaries. In this chapter, we focus on *secure multi-entity communication for* ***application-layer middleboxes***. Even among just application-layer middleboxes, security needs are potentially diverse—for example, intrusion detection systems and compression proxies behave very differently and trust relationships differ between an administrator-mandated virus scanner and an opt-in compression service—which suggests there may not be a single one-size-fits-all solution. Our first step toward answering these questions is to articulate the design space.

4.2.1 Design Space

TLS Security Properties. TLS currently provides the following properties in the two-party case. Clearly we want these properties in the multi-party case as well, but it turns out there are multiple ways to extend the two-party definitions to the multi-party case.

Data Secrecy and *Data Authentication.* Only the endpoints can read and write session data. By "write," we mean create, modify, delete, replay, or re-order messages. Furthermore, with a modern cipher suite, communication is *forward secret* (the compromise of a long-term private key does not help an attacker access previous sessions' data). To extend these properties beyond two parties, the following two questions arise.

Granularity of Data Access. `yes/no` `RW/RO/None` `func.` `crypto`
Do middleboxes have complete access to session data, or do they have some level of partial access? This could mean they can read/write some bytes but not others (e.g., HTTP headers but not HTTP bodies), as in mcTLS [177], or that they can perform a limited set of operations over encrypted data (e.g., search for patterns), as in BlindBox [209].

Definition of "Party." `machine` `program`
When a party is added to a session, is session data accessible to anyone with physical access to the machine, or only to the middlebox service software? This distinction becomes important when middleboxes are outsourced to third-party hardware (e.g., cloud providers or ISPs).

Entity Authentication. Each endpoint can verify that the other is operated by the expected entity by verifying that they possess a private key that a CA has certified belongs to that entity. To extend this property beyond two parties, the following question arises.

> *Definition of "Identity."* `owner` `code`
> When a party in a session verifies the "identity" of another party, what is it checking? That the machine is owned by the expected entity (e.g., this is a YouTube server)? That the machine is running the expected software and is correctly configured (e.g., Apache v2.4.25 with only strong TLS cipher suites enabled)? Both?

Other Security Properties. In the multi-party case, a number of new security properties arise.

Path Integrity. `yes` `no`
Does the protocol enforce that data must traverse middleboxes in a fixed order (and that they cannot be skipped)? Path order can impact security, especially when middleboxes perform filtering/sanitization functions.

Data Change *Secrecy.* `none` `value` `value + size`
Can the adversary learn anything about the communication by observing data before and after a middlebox? Protocols could offer no protection (adversary knows any time a middlebox makes a change), *value* protection (adversary does not learn when a middlebox changes a message so long as the size stays constant), or *value* + *size* protection (adversary does not learn about any changes).

Authorization. `0 endpts` `1 endpt` `both endpts` `endpts + mboxes`
Who gets to add a middlebox to a session (and decide what permissions it has)? Do both endpoints need to be made aware, so they can terminate the session if they do not approve? Only one? Should middleboxes be aware of other middleboxes?

Other Properties. Finally, there are a number of non-security properties that impact protocol deployability and usability.

Legacy Endpoints. `both upgrade` `1 legacy` `both legacy`
Do both endpoints need to be upgraded to a new protocol, or can one or both be legacy TLS endpoints?

In-Band Discovery. `yes` `yes + 1 RTT` `no`
Does the protocol allow endpoints to discover on-path middleboxes on-the-fly? If so, does adding discovered middleboxes add time to the handshake?

Computation. `arbitrary` `limited`
Does the protocol restrict what kinds of jobs middleboxes can perform? I.e., can they perform arbitrary computations on the data, or are they limited to a certain class of operations (e.g., pattern matching)?

		Custom Root Cert MitM *(current practice)*	Unilateral key generation *(mbTLS)*	Cooperative key generation *(mcTLS)*	Per-hop encryption keys *(mbTLS)*	Multiple encryption keys *(mcTLS)*	Searchable Encryption *(BlindBox)*
Data Granularity	`RW/RO/None`					✓	
	`func. crypto`						✓
Entity Auth.	`owner`	✗					
Path Integrity	`yes`				✓		
Legacy Endpoints	`1 legacy`	✓	✓	✗		✗	✗
	`both legacy`	✓	✗	✗	✗	✗	✗
Change Secrecy	`value`				✓		
Authorization	`1 endpoint`		✓	✗			
	`both endpoints`			✓			
Computation	`arbitrary`						✗

(✓ = enables; ✗ = prevents; no mark = no interaction)

Table 4.1: **Properties vs. Mechanisms.** Each column lists a mechanism a protocol might use and each row lists a property you might want the protocol to have. Many mechanisms enable some properties while at the same time preventing others.

4.2.2 Design Tradeoffs

Let us look at existing approaches in the context of this design space, highlighting how the mechanisms they introduce interact with the properties described above. It is often the case that a mechanism that provides a particular property along one dimension often eliminates options along another. Table 4.1 gives several examples of this; below we describe a few in detail.

TLS Interception with Custom Root Certificates [99, 132, 183] is the standard approach today. First, an administrator provisions clients with custom root certificates (this is easy in managed environments like corporate networks). Then, when the client opens a new connection, the middlebox intercepts, impersonates the intended server by fabricating a certificate for that domain, and opening a second connection to the server. Though this means both clients can be legacy TLS clients [*Legacy:* `both legacy`], it also makes it impossible for clients to authenticate the server [*Authentication:* ~~`owner`~~]—they must trust the middlebox to do so (trust which, in practice, is often misplaced [99]).

Multi-Context TLS (mcTLS) [177] offers *access control*—endpoints can restrict which parts of the data middleboxes can access and whether that access is read/write or read only [*Data access:* `RW/RO/None`]. It does this by encrypting different parts of the data with different keys and only giving middleboxes certain keys. This requires that both endpoints run mcTLS, precluding legacy endpoints, since a legacy TLS endpoint only knows what to do with one key [*Legacy:* ~~`1 legacy`~~ ~~`both legacy`~~]. Furthermore, each endpoint generates part of the key material for each of these keys, ensuring that a middlebox only gains access if both endpoints agree [*Authorization:* `both endpts`]. This also prevents legacy support.

BlindBox [209] offers *searchable encryption*—pattern-matching middleboxes like intrusion detection systems can operate directly on encrypted data [*Data access:* `func. crypto`]. But searchable encryption only works for pattern-matching; it cannot support other middleboxes, like compression proxies, that perform arbitrary computation [*Computation:* ~~`arbitrary`~~]. It also requires that both endpoints understand BlindBox's encryption scheme [*Legacy:* ~~`1 legacy`~~ ~~`both legacy`~~].

Middlebox TLS (mbTLS) *(this chapter).* We will soon see this for mbTLS too: for example, mbTLS uses a different symmetric key for each "hop" in the session, allowing mbTLS to provide path integrity [*Path integrity:* `yes`], but making it impossible to support *two* legacy endpoints [*Legacy:* ~~`both legacy`~~].

The takeaway is this: ***there is no one-size-fits-all solution for secure communication with application-layer middleboxes***. Each protocol here gives up desirable properties in order to provide others. Different properties, and therefore different protocols, will lend themselves best to different use cases.

4.3 Middlebox TLS

In this section, we present ***Middlebox TLS***, or ***mbTLS***, a protocol for secure multi-entity communication that lets endpoints establish a secure communication session that includes application-layer middleboxes. The solutions introduced in Section 4.2.2 fail to address some real-world needs, making them harder to deploy and reducing the incentive to do so in the first place. We saw in Section 4.2.2 that it is hard to build a super-protocol incorporating all the good features from Section 4.2.1; instead, we focus on the following three real-world needs:

(1) Protection for Outsourced Middleboxes: There is an increasing interest in deploying middleboxes in third party environments. This takes one of two forms. First, network functions can be outsourced to a cloud provider[1] that specializes in operating middleboxes, freeing network administrators from learning to operate specialized boxes and leveraging economy of scale to drive down costs [6, 31, 208]. Second, deploying middleboxes in client ISPs can help lower latency or bandwidth costs [14, 24, 33]. (For example, Google's Edge Network proxies connections using nodes in client ISPs [14].) In both cases, *the logical owner of the network function and the operator of the hardware it runs on are different.* Since the middlebox infrastructure may not be trusted, we

[1]This trend is encouraged by maturing technology for running middlebox applications on commodity hardware (NFV) [106, 114, 162, 205], including commercial offerings [7, 17, 20].

must **protect session data from the middlebox infrastructure** in addition to traditional network attackers.

(2) Legacy Interoperability: Protocols like BlindBox [209] and mcTLS [177] require both endpoints to be upgraded. Others require that at least the client be upgraded [156, 159, 164], meaning servers cannot include middleboxes in a session with a legacy client. Realistically, however, it is not an option to wait until every client in the Internet is upgraded; this is particularly true given that as many as 10% of HTTPS connections are *already* intercepted [99]. Therefore, it is crucial to **support legacy endpoints**.

(3) In-Band Discovery: This is important for practical deployment in the use cases we target. For example, suppose a service provider places proxies in edge ISPs. Directing clients to connect to their local proxy using DNS (1) is an unnecessary configuration burden and (2) is brittle, since clients can use non-local DNS resolvers like OpenDNS. Another example is guest networks where administrators cannot feasibly configure every client device that joins (nor would those users want them to).

4.3.1 Threat Model

Parties. We identify six distinct parties and label each as "trusted" or "untrusted," where trusted means that party is authorized to access session data.

Client (C) [trusted]: The user, their machine, and the software they run (e.g., a web browser). We assume any other software running on the machine is trusted (i.e., misbehavior by this software is out of scope).

Service Provider (S) [trusted]: The company providing the online service, its servers, and the software it runs (e.g., a web server). As with the client, we do not consider attacks by other software running on the company's servers or by malicious employees.

Third Parties (TP) [untrusted]: Anyone else with access to network traffic, such as ISPs or coffee shop Wi-Fi sniffers.

Middlebox Software (MS) [trusted]: The middlebox software that processes session data.

Middlebox Service Provider (MSP) [trusted]: The entity offering the middlebox service.

Middlebox Infrastructure Provider (MIP) [untrusted]: The entity providing the hardware on which the *MS* runs.

The *MIP* could be the *MSP* itself or a third party such as a customer ISP or a dedicated cloud middlebox service, in which case we assume this company, its employees, its hardware, and any other software running on its machines are *not* trusted. For example, suppose Google implements their Flywheel compression proxy [37] using Apache httpd running on Amazon EC2. In this case, *MS* = Apache httpd, *MSP* = Google, and *MIP* = Amazon. By distinguishing between *MS* and *MIP*, we will require mbTLS to permit only middlebox software to access session data. [*Party:* `program`]

Adversary Capabilities. We assume an active, global adversary that can observe and control any untrusted part of the system. In the network, the adversary can observe, modify, or drop

any packet and inject new ones. On the middlebox infrastructure, the adversary has complete access to all hardware (e.g., it can read and manipulate memory) and software (e.g., it can execute arbitrary code, including privileged code like a malicious OS). This includes the ability to modify or replace middlebox code sent by the MSP to be executed by the MIP. We assume the adversary is computationally bounded (i.e., cannot break standard cryptographic primitives) and cannot compromise trusted computing hardware (in our case, Intel SGX-enabled CPUs). Side channel attacks (e.g., based on traffic or cache access patterns), exploitable flaws in middlebox software, and denial of service are out of scope.

4.3.2 mbTLS Properties

Based on the design requirements above, we define "secure" for mbTLS by the following four security properties. "The adversary" means any party marked untrusted in Section 4.3.1.

P1 Data Secrecy. **P1A** The adversary must not be able to read session data. **P1B** Communication should be *forward secret* (the compromise of a long-term private key does not help an attacker access previous sessions' data). **P1C** The adversary should learn nothing more from observing ciphertext than it would if each "hop" were its own, independent TLS connection. [*Change secrecy:* `value`]

P2 Data Authentication. The adversary must not be able to modify, delete, or inject session data. This includes replaying or re-ordering data.

P3 Entity Authentication. Endpoints must be able to verify they are talking to the "right thing." This encompasses two subtly intertwined properties. **P3A** Each endpoint can verify that the other endpoint is operated by the expected entity and that each *MS* is operated by the expected *MSP* (e.g., a YouTube server). [*Identity:* `owner`] **P3B** Each endpoint can verify that the other endpoint and each *MS* is running the expected software and that it is correctly configured (e.g., Apache v2.4.25 with only strong TLS cipher suites enabled). [*Identity:* `code`]

P4 Path Integrity. The endpoints fix an ordered path of middleboxes for a session. It must not be possible for any other entity (including a middlebox) to cause session data to be processed by middleboxes in a different order (including skipping a middlebox). Note that all client-side middleboxes are required to come before all server-side middleboxes (see Section 4.4.2). [*Path integrity:* `yes`]

In addition to these security properties, we also have the following performance and functionality-related goals:

P5 Legacy Interoperability. mbTLS should work with legacy TLS endpoints (client or server) so long as one of the endpoints has been upgraded. [*Legacy endpoints:* `1 legacy`]

P6 In-Band Discovery. mbTLS should discover on-path middleboxes during session setup. [*Discovery:* `yes`]

P7 Minimal Overhead. mbTLS should introduce as little overhead (compared to TLS) as possible. Importantly, mbTLS should not add any round trips to the TLS handshake.

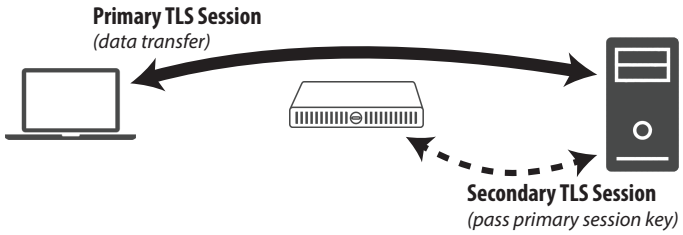


Figure 4.1: **Naïve Approach.** Establish a TLS session end-to-end and pass the session key to the middlebox over a secondary TLS session.

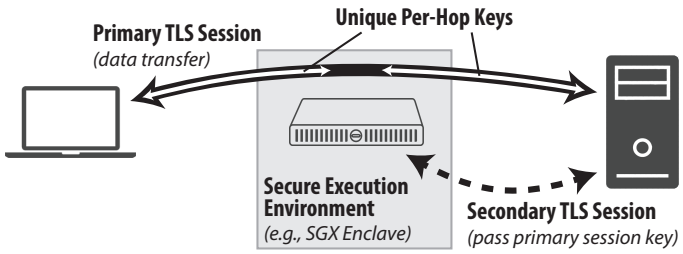


Figure 4.2: **mbTLS Approach.** Generate unique keys for each "hop" and run middleboxes in secure execution environments.

4.3.3 Design Overview

Since TLS already provides many of the properties we want, one simple approach is the following: establish a regular TLS session between the client and the server, then pass the session keys to the middleboxes over separate, secondary TLS sessions (Figure 4.1) [187]. This provides many of the security properties we want: data is encrypted and integrity-protected against changes from third parties, the communication is forward secret if a forward secure cipher suite is used, and the endpoints can verify one another's identify using certificates.

However, using TLS in this way is insufficient in our threat model for three reasons: (1) it has no mechanism to provide path integrity since it was designed for two parties **P4**; (2) the same key is used for encryption on each "hop" in the session, making it simple for adversaries to compare records entering and leaving a middlebox to see if they changed **P1C**; and (3) the infrastructure provider can access session data in memory **P1A**, access key material in memory and use it to forge MACs **P2**, and potentially run software other than what was provided by the *MSP* **P3B**.

We address these insufficiencies by introducing the following features (Figure 4.2), and we call the result ***Middlebox TLS (mbTLS)***.

- **In-Band Middlebox Discovery.** As long as one of the endpoints supports mbTLS, middleboxes can announce themselves and join the session (with endpoint approval) during the primary TLS handshake (**P6**).
- **Secure Execution Environments.** Middleboxes can optionally run in a secure execution environment, like an Intel SGX enclave, to protect session data and keys from an untrusted *MIP* (**P1A**, **P2**) and to allow endpoints to verify the software identity of the *MS* (**P3B**).
- **Unique Per-Hop Keys.** Each "hop" uses its own symmetric keys for protecting session data. This prevents adversaries from delivering records to an out-of-sequence middlebox (**P4**) and

makes it impossible to tell when a middlebox forwards data without changing it (**P1C**).

An Aside: Trusted Computing and SGX. Some features of mbTLS rely on trusted computing technology, like Intel's Software Guard Extensions (SGX) [39, 129, 165]. In particular, mbTLS uses two features provided by SGX—secure execution environments and remote attestation—though any trusted computing technology that offers these features, like Microsoft's Virtual Secure Mode (VSM) [45] or ARM TrustZone [5] would work as well. (Other technologies, like ARM TrustZone [5], offer similar functionality, but provide slightly different security guarantees.) We briefly describe these features now; if you are familiar with SGX, skip ahead to Section 4.3.4.

Secure Execution Environment. SGX allows applications to run code inside a secure environment called an *enclave*. An enclave is a region of protected memory containing program code and data; before cache lines are moved to DRAM, they are encrypted and integrity-protected by the CPU. As long as the CPU has not been physically compromised, even malicious hardware or privileged software cannot access or modify enclave memory.

Remote Attestation. SGX can provide code running in an enclave with a special message, signed by the CPU, called an *attestation*, that proves to remote parties that the code in question is indeed running in an enclave on a genuine Intel CPU. The attestation includes a cryptographic hash of initial state of the enclave code and data pages (so the remote verifier can see that the expected code is running) as well as custom data provided by the enclave application (we use this to integrate attestation with the TLS handshake).

4.3.4 The mbTLS Protocol

At a high level, the endpoints do a standard TLS handshake, establishing a *primary TLS session*, which will eventually be used for data transfer. Each endpoint adds zero or more middleboxes to a session, which we refer to as *client-side* and *server-side* middleboxes and can be known a priori or discovered during the handshake (**P6**). Each endpoint has no knowledge of the other's middleboxes (or if it has any at all) which means an mbTLS endpoint can inter-operate with legacy TLS endpoints (**P5**). The endpoints simultaneously establish a *secondary TLS session* with each of their middleboxes. Once an endpoint has a secure channel to a middlebox (which can include verifying that the middlebox software is running in a secure execution environment), it sends the middlebox the key material it needs to join the primary end-to-end session.

Control Messaging. mbTLS uses the same per-hop TCP connections for the primary and secondary handshakes. Compared to opening secondary TCP connections, this reduces overhead (**P7**) by (1) reducing TCP state on both middleboxes and endpoints, (2) keeps all handshake messages on the same path since the overall handshake is only as fast as the slowest, least reliable path, and (3) keeps client-side middleboxes discovery from adding a round trip (see below). We introduce a new TLS record type (`Encapsulated`) to wrap secondary TLS records between a middlebox and its endpoint. These records consist of an outer TLS record header followed by a one byte *subchannel ID* and the encapsulated record. For details on mbTLS message formats, see [18].

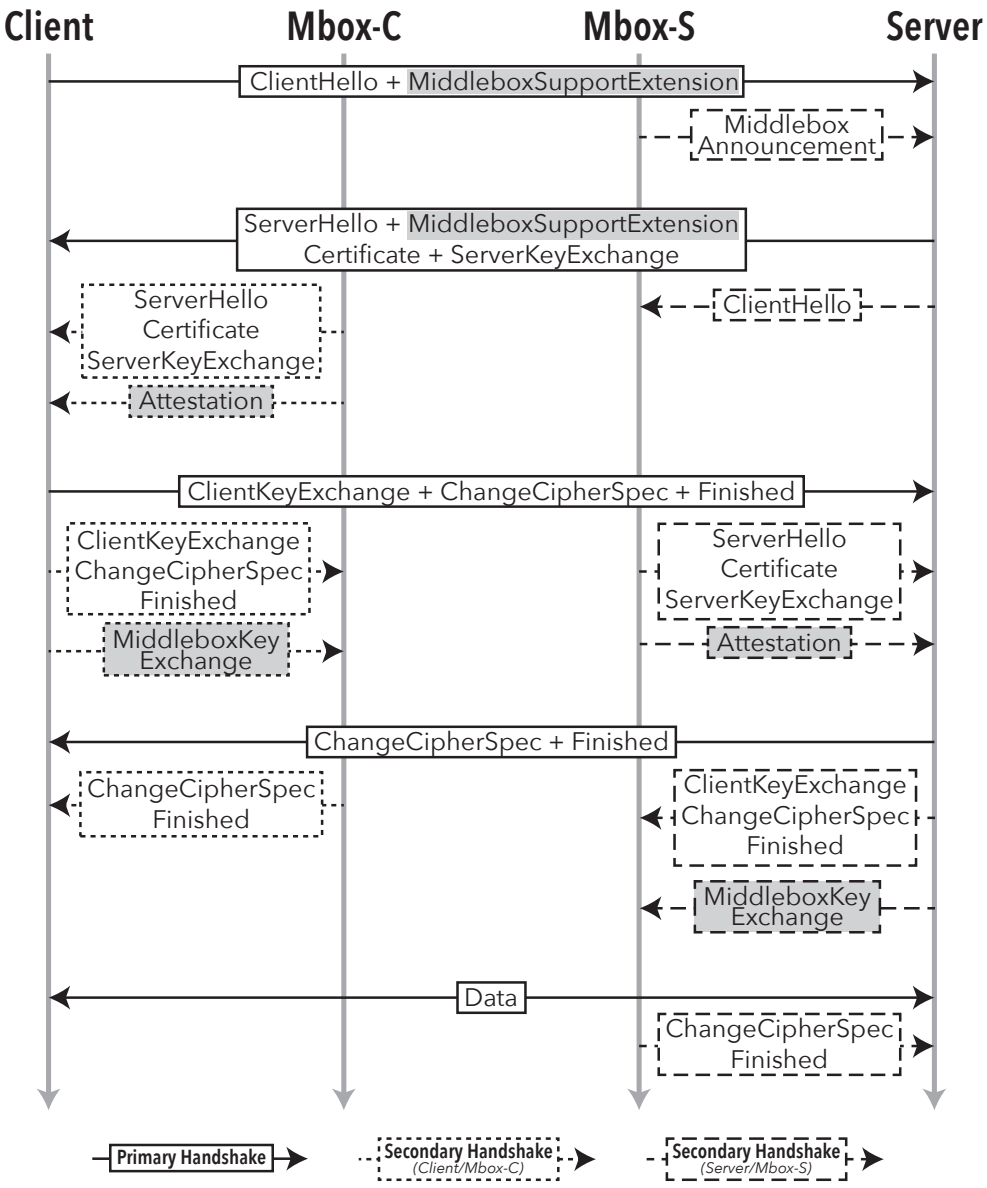


Figure 4.3: **mbTLS Handshake.** Note that it consists of multiple standard TLS handshakes, interleaved, with a few additional messages (shaded).

Client-Side Middlebox Discovery. mbTLS allows clients to include both middleboxes known a priori (e.g., configured by a user or announced via DNS, DHCP, or PDP/PDN) and those discovered during session establishment (on the default routing path) **P6**. To inform on-path middleboxes that the client supports mbTLS, the primary `ClientHello` includes a new `MiddleboxSupport` TLS extension. When it sees the extension, the middlebox (1) forwards the `ClientHello` onward toward the server and (2) begins its own secondary handshake with the client. In this secondary handshake, the middlebox plays the role of the server. The original, primary `ClientHello` serves double-duty as the `ClientHello` for the secondary handshake as well; the middlebox responds directly with a `ServerHello` (this is to avoid an extra round trip **P7**). However, in all computations, both the client and the middlebox use $PRF(\texttt{ClientRandom}||\texttt{MiddleboxRandom})$ in place of the original `ClientRandom` used in the primary handshake.

There may be multiple client-side middleboxes. Secondary handshake messages are sent in `Encapsulated` records, each middlebox with its own subchannel ID. Middleboxes wait until they see the primary `ServerHello`, buffer it, assign themselves the next available subchannel ID, inject their own secondary `ServerHello` into the data stream using that ID, and finally forward the primary `ServerHello`. This process ensures that each middlebox gets a unique subchannel ID with minimal coordination.

Server-Side Middlebox Discovery. Server-side middleboxes can also be pre-arranged (e.g., via DNS) or discovered on the fly (**P6**). Discovery is slightly trickier in the server-side case, however. Unlike the client, the server does not announce mbTLS support using the `MiddleboxSupport` extension for two reasons: first, the TLS spec forbids the server from including an extension in the `ServerHello` that the client did not include in the `ClientHello` [94]; relying on a `MiddleboxSupport` extension for the server would fail if the client does not also support mbTLS. Second, even if this were possible, if server-side middleboxes waited to announce their presence until after the server's `ServerHello`, the middlebox-server handshake would finish after the primary handshake, lengthening the overall handshake process to more than two RTTs (against **P7**).

Instead, server-side middleboxes optimistically announce themselves with a new `MiddleboxAnnouncement` message *before* they know if the server supports mbTLS. If it does not, then depending on its TLS implementation, it will either ignore the `MiddleboxAnnouncement` and the handshake will proceed without the middlebox, or the handshake will fail. (In either case, the middlebox will cache this information and not announce itself to this server again.) If the handshake fails, the client will need to retry. There is a potential danger that client software might interpret this to mean the server is running an out-of-date TLS stack and retry using an older version of TLS. We verified that Chrome and Firefox do *not* exhibit this behavior; Chrome will try a second TLS 1.2 handshake, and Firefox will not retry at all (meaning the user will need to manually click "Refresh"). Furthermore, in practice, we expect server-side middleboxes and servers will typically be under the same administrative control, in which case the middleboxes know that the server supports mbTLS. Like the client-side middleboxes, server-side middleboxes assign themselves unused subchannel IDs when they send their `MiddleboxAnnouncement` messages.

Secure Environment Attestation. We have extended the TLS 1.2 handshake (see Figure 4.4) to optionally include a remote attestation (in addition to the standard certificate check), which

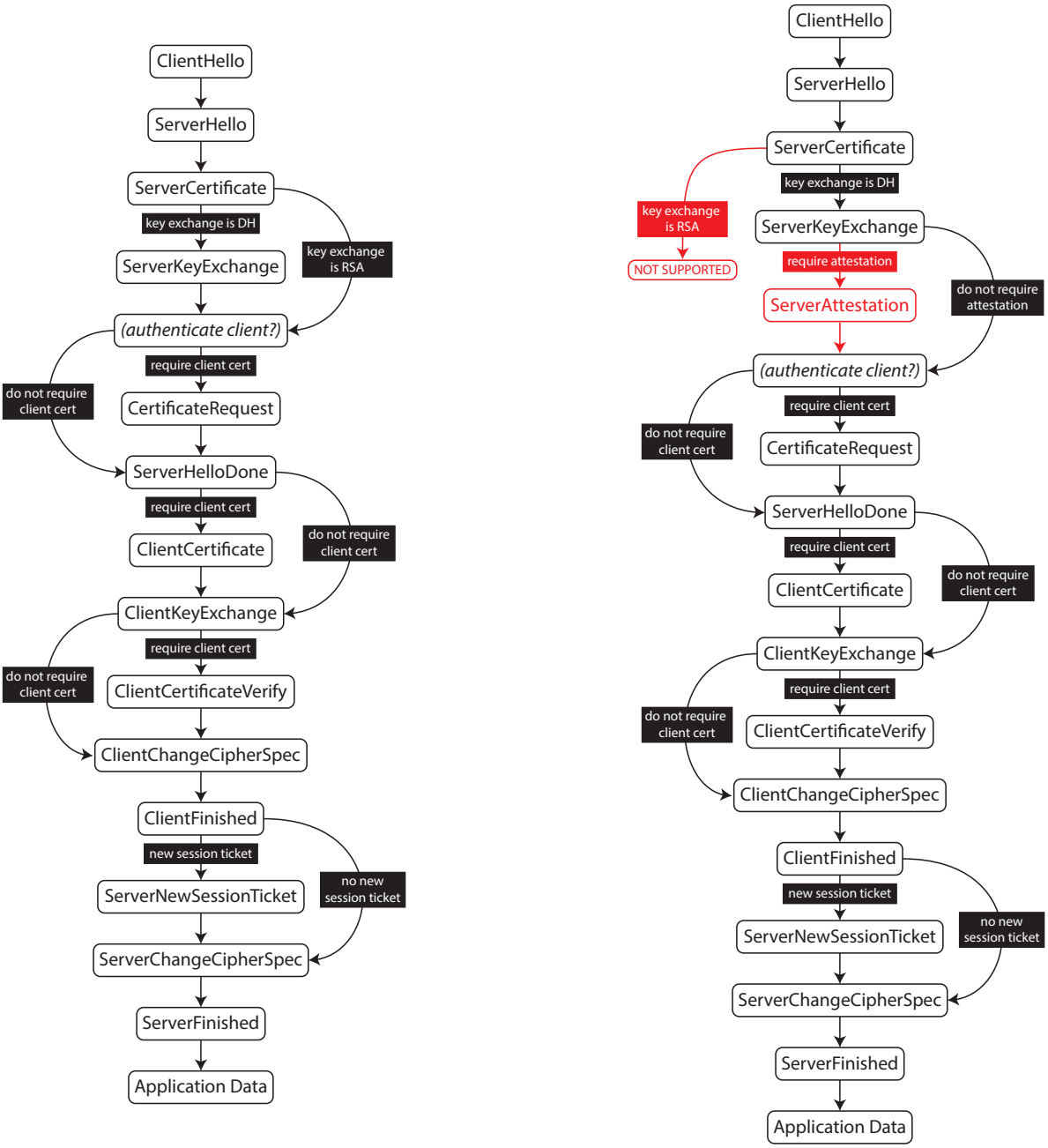


Figure 4.4: **Updated State Machine.** The TLS 1.2 state machine (left) and our modification to support remote attestation (right). Changes to the TLS state machine should be made carefully; we argue this change is a small one, and easy to reason about.

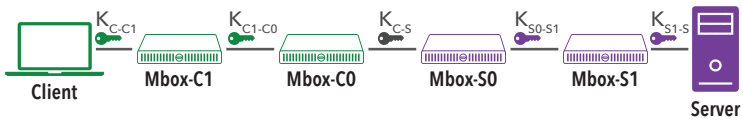


Figure 4.5: **Unique Per-Hop Keys.** Each hop encrypts and MAC-protects data with a different key—the client generates keys for the client-side hops (K_{C-C_1} and $K_{C_1-C_0}$), the server generates keys for the server-side hops ($K_{S_0-S_1}$ and K_{S_1-S}), and the primary session key (K_{C-S}) bridges the sides.

the endpoints may use here to verify that (1) the secondary (or primary) TLS sessions terminate inside a secure execution environment (**P1A**, **P2**) and (2) the middlebox is running the expected software in the expected configuration (**P3B**). The goal is to convince the endpoint that *only the middlebox application running in the enclave knows the TLS session key being established*. The main idea is the following: since the attestation includes the identity of the code, and we assume the code (application + mbTLS library) has been inspected and is trusted, then if the code tells us that it generated the secret key material for this handshake and did not export it, then we can trust it. The challenge becomes identifying "this handshake"—how can the endpoint be sure an adversary is not replaying an old attestation from a different handshake?

This means, in addition to the code identity, the attestation must include some kind of *handshake identifier* (e.g., SGX allows attestations to include 64 bytes of arbitrary "user data"). A good handshake identifier should be something that (1) is present in every handshake (so, not the session ID, which the server can choose not to support), (2) will not normally repeat in future handshakes, and (3) cannot be forced to repeat by an attacker (so, not the client random). Good candidates include anything based on the ephemeral keys exchanged in the handshake. The pre-master secret, or anything derived from it, would be a good choice, except this is only known to the middlebox after receiving the `ClientKeyExchange` from the endpoint. If we wait this long to send the attestation, we delay the overall end-to-end handshake. Instead, we base the handshake identifier on just the middlebox's key material (our implementation uses a hash of the middlebox's public ephemeral Diffie Hellman key). It is okay that these are public because they do not repeat normally and an attacker cannot force them to. This requires that the server use a key exchange method with an ephemeral public key (since a fixed public key will be the same in each handshake), but using ephemeral keys for forward secrecy is standard best practice anyway.

Unique Per-Hop Keys. At the end of a mbTLS handshake, the session looks like Figure 4.5. After finishing the secondary handshakes with its middleboxes, each endpoint generates a symmetric key for each hop on its side of the connection (e.g., the client generates K_{C-C_1} and $K_{C_1-C_0}$ in the figure). This prevents an adversary from causing records to skip a middlebox or traverse the middleboxes out-of-order (**P4**) and also prevents eavesdroppers from detecting whether or not a middlebox modified a record (**P1C**). Endpoints distribute these keys to their middleboxes in `MiddleboxKeyExchange` records encrypted using the secondary connection keys, which, just like the secondary handshake messages, are sent in `Encapsulated` records in the data stream. The session key established as a result of the primary handshake, K_{C-S}, serves as a "bridge" between the client-side and server-side middleboxes (or between a middlebox and a legacy endpoint **P5**).

Security Property	Threat	Defense (TLS)	Defense (mbTLS)
Data Secrecy	**P1A** Data read on-the-wire by *TP* or *MIP*	Encryption	Encryption*
	P1A Data read in *MS* application memory by *MIP*	—	Secure Execution Environment
	P1C *TP* compares record entering and leaving *MS* to see if it was modified	—	Unique Per-Hop Keys
Data Authentication	**P2** Records dropped, injected, or modified on-the-wire	MACs	MACs*
	P2 Data deleted, injected, or modified in RAM by *MIP*	—	Secure Execution Environment
Entity Authentication	**P3A** *C* establishes key with wrong software on *S*	Certificate	Certificate
	P3A *C* establishes key with software on hardware operated by someone other than *S*	Certificate	Certificate
	P3B *C* or *S* establishes key with wrong *MS* software	—	Remote Attestation
	P3A *C* or *S* establishes key with *MS* software operated by someone other than *MSP*	—	Certificate*
Path Integrity	**P4** Records passed to middleboxes in the wrong order	—	Unique Per-Hop Keys

Table 4.2: **Threats and Defenses.** How mbTLS defends against concrete threats to our core security properties. For comparison, we include TLS where applicable. An asterisk indicates that defense also relies on the secure environment to safeguard the session key.

4.3.5 Discussion

Session Resumption. mbTLS fully supports both ID-based and ticket-based session resumption. Each sub-handshake (the primary handshake and the secondary handshakes) simply does a standard abbreviated handshake; the only minor difference is that the session tickets for middleboxes should contain the session keys for the end-to-end session (in addition to the key for the endpoint-middlebox sub-session). A new attestation is not required, because only the enclave knows the key needed to decrypt the session ticket. A client that wishes to resume a session stores a session ID or ticket for the server and each client-side middlebox. If the server also uses mbTLS, it can either cache the session IDs/tickets for its middleboxes or ask the client cache them and send them in its `ClientHello`.

TLS 1.3. TLS 1.3 [197] significantly changes the TLS handshake compared to TLS 1.2 and earlier, shortening it from two round trips to just one. With minor modifications, mbTLS's handshake can be adapted to TLS 1.3. There is one caveat: when client-side middleboxes are present, data sent by the server in the same flight as the server `Finished` could be delayed, in the worst case, up to one round trip. In most cases, however, clients send application data first; in these cases, there is no issue.

4.4 Security Analysis

4.4.1 Core Security Properties

We now revisit each security property from Section 4.3.2, arguing why mbTLS provides it. Table 4.2 summarizes the concrete threats we address, how TLS defends against them in the two-party case, and how mbTLS defends against them in the multi-party case.

P1 Data Secrecy.

P1A *The adversary must not be able to read session data.* Decrypting session data requires access to one of the symmetric keys shown in Figure 4.5. The bridge key, K_{C-S}, is established during the end-to-end client-server TLS handshake in which the endpoints verify one another's certificates. Next, this key and the rest of the session keys (e.g., K_{C-C1}, K_{C1-Co}, etc.) are transferred to the middleboxes over individual secondary TLS connections; importantly, these secondary connections terminate inside the SGX enclave, meaning the *MIP* cannot access the secondary session's key in memory, so only the *MS* (and not the *MIP*) learns the primary session keys. Remote attestation proves to a middlebox's endpoint that the *MS* is truly running in the secure environment.

P1B *Communication should be forward secret.* The bridge key (K_{C-S}) is the result of the (standard) primary TLS handshake, so if the primary handshake is forward secure, so is K_{C-S}. The other session keys (e.g., K_{C-Co}, K_{Co-C1}, etc.) are generated fresh for each session and sent to the middleboxes over (standard) secondary TLS connections. Therefore, if these secondary handshakes are forward secure, so are the non-bridge session keys.

P1C *The adversary should learn nothing more from observing ciphertext than it would if each hop were its own, independent TLS connection.* Since each hop uses its own independent encryption and MAC keys, after the handshake each hop effectively operates like its own TLS connection. In particular, this prevents an adversary from learning whether or not a middlebox modified a record (though it can still see the sizes and timings of each record, including whether a middlebox increased or decreased the size of the data).

P2 Data Authentication. *The adversary must not be able to modify, delete, or inject session data.* Each record carries a *message authentication code (MAC)*, a small tag generated using the session key that identifies a piece of data. Unauthorized changes can be detected if the MAC does not match the data. Since only the endpoints and each *MS* know the session keys (see P1A), only these entities can modify or create records.

P3 Entity Authentication.

P3A *Each endpoint can verify that the other endpoint is operated by the expected entity and that each MS is operated by the expected MSP.* First, the client and server can require one another's certificate in the primary handshake (though typically client authentication happens at the application layer). A certificate binds the server's public key to its identity, and that public key is used in the primary handshake to negotiate the shared bridge key, so after a successful handshake, the client is assured that any data encrypted with that bridge key can only be decrypted by the expected service provider (or middleboxes it chose to add to the session). Second, endpoints can also require certificates from middleboxes. Since the private key corresponding to the certificate is stored in the enclave, inaccessible by the *MIP* (and remote attestation proves that this is the case), the endpoint is convinced it it talking to software supplied and configured by the expected *MSP*.

P3B *Each endpoint can verify that the other endpoint and each MS is running the expected software and that it is correctly configured.* Since our threat model assumes that the *SP* and all software running on its server is trusted, and in **P3A** we verified that the server possesses the *SP*'s private key, the client trusts that the machine is properly configured with the expected application software.

The same logic applies to the middleboxes, with the additional step that the remote attestation convinces the endpoint that the MS is safely isolated in the secure execution environment.

P4 Path Integrity. *Each endpoint picks an order for its middleboxes. It must not be possible for any other entity (including the other endpoint or any middlebox) to cause session data to be processed by middleboxes in a different order.* This follows from the fact that mbTLS uses a fresh key for each hop. Suppose an adversary sniffs a record from the $C1 - Co$ link in Figure 4.5 and tries to insert it on the $So - S1$ link (thereby skipping middleboxes Co and So). The record will be encrypted and MAC'd with K_{C1-Co}, but $C1$ expects data secured with K_{S1-So}, so the MAC check will fail and the record will be discarded. (Note, that an *endpoint* can inject, delete, or modify data anywhere in its portion of the path because it knows all the session keys on its side. We discuss potential security implications below.)

4.4.2 Other Security Properties

Endpoint Isolation. *Endpoints can only authenticate their own middleboxes, not those added by the other endpoint.* In fact, an endpoint likely does not even know about the other side's middleboxes. This follows from the way keys are generated and distributed. Checking a certificate or an attestation is only meaningful if the public key in the certificate is used for key exchange (then you trust that only the entity associated with that public key can decrypt what you send with the new symmetric key). Since endpoints don't do a KE with the other side's middleboxes, they have no means of authenticating one another, even if they exchanged certificates/attestations. This limitation seems reasonable; since the endpoints presumably trust one another or they would not be communicating to begin with, it is natural to trust the other endpoint to properly authenticate any middleboxes it adds to the session.

Path Flexibility. *It is not possible to interleave client-side and server-side middleboxes.* To support this, the endpoints would need to coordinate to generate/distribute keys to the interleaved portion of the path. This means (1) extra work for endpoints and (2) the endpoints would need to know about (some of) one another's middleboxes. This would also mean that one endpoint could modify/inject traffic *after* the other endpoint's middleboxes, which could be a security problem if one of those middleboxes does some kind of filtering or sanitization.

Untrusted MSPs. *mbTLS can provide guarantees even when the service provider is untrusted.* In our threat model, both the SP and the MSPs are trusted. However, even in a more pessimistic threat model where they are *untrusted*, remote attestation can still provide **P1**, **P2**, **P3**, and **P4**, since the attestation identifies the code running in the secure environment. This relies on two big assumptions: (1) that software is known to "behave well" (e.g., does not export session data outside the enclave), which is a difficult problem, and (2) that the client knows a hash of this "known good" software. For example, a client could connect to an untrusted Web proxy if the software is open source and has been publicly verified to keep session data confidential, even if the client trusts neither the company operating the service nor the infrastructure it runs on.

Middlebox State Poisoning. *It is not safe to use mbTLS with **client-side** middleboxes that keep **global state**.* Since endpoints know the keys for each hop on their side of the connection, a malicious client can read and/or modify data on any of these hops without its middleboxes knowing. This is a problem when a middlebox that shares state across multiple clients, like a Web cache. A client with access to a link between the cache and the server could (1) request a page, (2) drop the server's response, and (3) inject its own response, thereby poisoning the cache for other clients.

One possible solution is to alter the handshake protocol so that middleboxes establish keys with their neighbors rather than endpoints generating and distributing session keys; this means each party only knows the key(s) for the hop(s) adjacent to it. The downside is the client has lost the ability to verify the server's certificate and establish a session key using the public key in that certificate. Instead, the client must trust its middleboxes to authenticate the server. This may be reasonable, since the SGX attestation should convince the client that the middlebox is running software that will do so, but we did not take this approach in mbTLS because, where possible, we prefer to rely on cryptography, since relying on SGX also means relying on the correctness of the protocol library code.

Bypassing "Filter" Middleboxes. At first glance it appears that the fact endpoints know all the session keys on their side opens another attack: if a middlebox performs some kind of filtering function (e.g., a virus scanner, parental filter, or data exfiltration detector mandated by an administrator), this means the endpoint has the keys to access incoming data *before* it is filtered or inject outbound data afterward. However, if an endpoint is capable of reading or writing data "on the other side of" of the filter (i.e., physically retrieve/inject packets from/into the network beyond the middlebox), then the filter was useless to begin with, so mbTLS does not enable new attacks.

4.5 Evaluation

We evaluate four critical aspects of mbTLS. First, our security analysis argues that mbTLS is **secure** (Section 4.4). Second, with a series of real-world experiments, we show that mbTLS is **immediately deployable** (Section 4.5.1). Third, we show mbTLS imposes **reasonable CPU overhead** for servers wishing to deploy it (and reduces it for middleboxes) (Section 4.5.2). Finally, we show that **SGX applications can support network I/O heavy workloads** (Section 4.5.3).

Prototype Implementation. We implemented mbTLS in OpenSSL (v1.1.1-dev) using the Intel SGX SDK for Windows (v1.7). Our prototype currently supports any cipher suites using `DHE` or `ECDHE` for key exchange and `AES256-GCM` for bulk encryption. Recall that, if mbTLS is used with SGX, our attestation procedure requires an ephemeral key exchange (Section 4.3.4). However, there is no reason we could not support encryption algorithms other than `AES256-GCM`. We also provide a support library for running our mbTLS implementation inside an SGX enclave. Our support library implements 8 libc functions directly in the enclave (3 of which are only used for debugging and can be removed in production builds) and exits the enclave for another 7 libc functions (4 of which are for debugging). The middlebox in the following experiments is a simple HTTP proxy that performs HTTP header insertion.

Testbed. Our local testbed comprises four servers running Windows Server 2016, with SGX-

Network Type	# Sites	Network Type	# Sites
Enterprise	6	*University*	11
Residential	34	*Public*	1
Mobile	2	*Hosting*	56
Colocation Services	35	*Data Center*	19
Uncategorized	77	**Total**	241

Table 4.3: **Handshake Viability.** Number of distinct sites from which we performed mbTLS handshakes to our test server, broken down by network type. All handshakes were successful.

enabled Intel Core i7 6700 processors and an SGX-enabled motherboard. These are connected through Mellanox ConnectX-3 40Gbps network cards to a local Arista 7050X switch.

4.5.1 Deployability

We test two things through real-world deployments: (1) Do firewalls or traffic normalizers in the public Internet block mbTLS connections? (2) Can mbTLS interoperate with legacy endpoints?

Handshake Viability. Since mbTLS introduces new TLS extensions (`MiddleboxSupport`) and record types (`Encapsulated` and `MiddleboxAnnouncement`), we verify that existing filters, like firewalls, traffic normalizers, or IDSes, do not drop our handshakes. To do so, we connect to a middlebox and server running in Azure from clients located in various networks around the world. The middlebox is configured to be a client-side middlebox, so the new message types traverse the networks between the client and the data center. To test a diverse set of client networks, we do two things. First, we run our client through TOR, using 550 exit nodes located in 46 countries across 214 ASes. Using whois data, we categorized the networks by type. We chose not to use platforms such as PlanetLab as these are mainly placed in university networks, which are typically not heavily filtered. Second, we manually connect from different types of networks to fill in gaps in the Tor experiment (namely public, mobile, and data center networks). Table 4.3 shows a breakdown of the distinct client networks from which we initiated a handshake. All handshakes were successful.

Legacy Interoperability. To demonstrate that mbTLS can communicate with legacy endpoints, we use a version of curl [8] modified to use mbTLS to download the root HTML document for the top 500 Alexa sites that support HTTPS via our SOCKS HTTP proxy running in Azure. Only 385 sites in the Alexa top 500 support HTTPS; we successfully connected to 308 of these. Of the 77 that failed, 19 had invalid or expired certificates. Another 40 did not support `AES256-GCM`, the only encryption algorithm currently supported by our prototype (note that this is a limitation of our prototype, not the protocol). Another 13 failed due to redirects our SOCKS implementation did not properly handle. We are currently debugging the remaining 5, though we are confident the failures were not due to a protocol flaw.

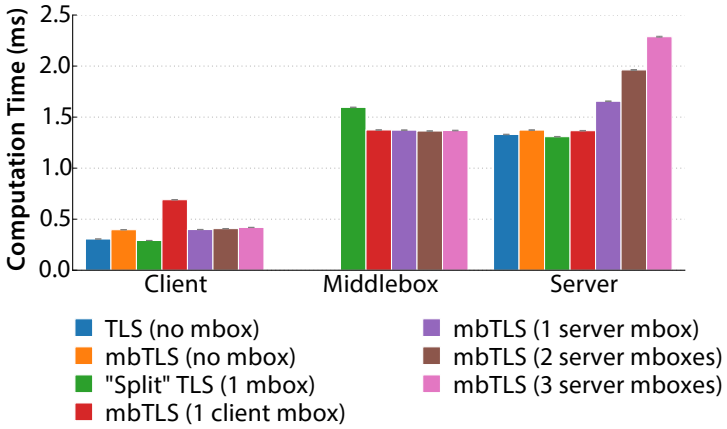


Figure 4.6: **Handshake CPU Microbenchmarks.** Each bar shows the time spent executing a single handshake (not including waiting for network I/O). Each bar is the mean of 1000 trials; error bars show a 95% confidence interval of the mean.

4.5.2 Performance Overhead

mbTLS does not modify the TLS record layer, so it has no impact on data transfer performance. On the other hand, it *does* change the handshake, so we (1) investigate the computational expense of performing a handshake (a concern for servers and middleboxes, which need to handle many connections), and (2) empirically verify that mbTLS does not increase session setup latency.

Handshake CPU Microbenchmarks. We measured the time it costs clients, middleboxes, and servers to perform TLS and mbTLS handshakes. As Figure 4.6 shows, without a middlebox, the TLS and mbTLS times are close (we suspect the difference is inefficiency in our implementation, not any fundamental property of the protocol). Second, for the middlebox, a mbTLS handshake is *cheaper*, because the middlebox only performs one TLS handshake (as opposed to two in Split TLS). Finally, the server's load is not impacted by client-side middleboxes and increases linearly with the number of server-side middleboxes. Note, however, that each server-side middlebox only adds approximately 20% of the original, no-middlebox handshake time; this is because, for each middlebox, the server performs one additional *client* TLS handshake, which is cheaper than a server handshake. (Key exchange was `ECDHE-RSA`; results were similar for `DHE-RSA`. Machine specs: Intel i7-6700K CPU at 4 GHz; 16 GB RAM; Windows 10.)

Handshake Latency. To confirm that our handshake protocol does not inflate session setup in practice (it should not, since it maintains the same four-flight "shape" as TLS), we perform several handshakes across data centers in Microsoft Azure. We deploy VMs in four regions (Australia, US West, US East, and UK) and test all permutations of a client-middlebox-server path across them (with no two VMs in the same DC). In each test, we compare the time to fetch a small object using mbTLS and TLS. For regular TLS, the middlebox simply relays packets, i.e., it does not perform split TLS—this is the worst-possible case to compare mbTLS against since the middlebox performs no handshake operations. Figure 4.7 summarizes the results, broken into handshake time and data transfer time. Each bar is the mean of 100 trials; error bars show a 95% confidence interval.

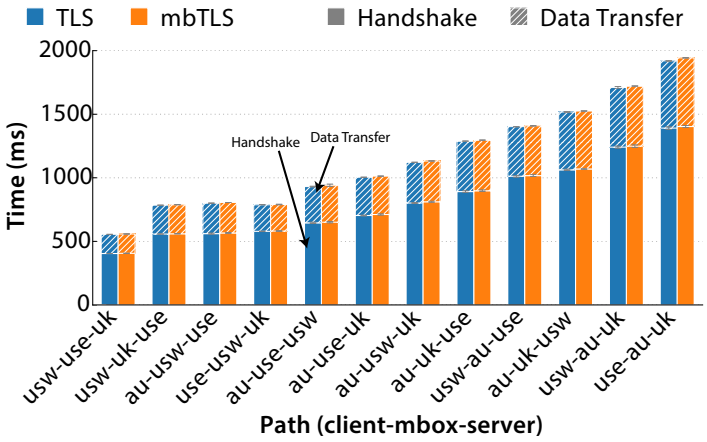


Figure 4.7: **mbTLS vs. TLS Latency.** Time to fetch a small object across various paths between data centers.

We observe that mbTLS increases the handshake latency on average by 0.7% (1.2% in the worst case—a 10 ms increase out of 800 ms). This is likely due simply to handshake computations on the middlebox.

4.5.3 Network I/O in SGX

Finally, we investigate network I/O preformance from the enclave. SGX imposes restrictions on what enclave code can do. Since only the CPU is trusted, interaction with the outside world is not permitted by default (notably, system calls are not permitted, since the OS is untrusted). When an enclave thread needs to make a system call, there are two high-level strategies: (1) it copies the arguments into unprotected memory, exits the enclave, executes the call, re-enters the enclave, and copies the result back into enclave memory (this boundary-crossing incurs a performance penalty); or (2) it places a request in a shared queue and another thread outside the enclave executes the call passes the result back into the enclave via a response queue. To borrow terminology from SCONE [44], these are *synchronous* and *asynchronous* system calls, respectively.

In a microbenchmark of repeated `pwrite()`s, SCONE found that, for small buffer sizes, asynchronous calls can be up to an order of magnitude faster. However, here we are concerned specifically with `send()`s and `recv()`s, so we performed a small experiment to test the enclave's impact on throughput. We configured four machines in our lab to be a middlebox, a server and two clients. The clients send a stream of random bytes, sent in encrypted chunks whose size we vary. The middlebox is configured with one of four behaviors: it either simply forwards the (encrypted) data to the server or it decrypts and re-encrypts it before forwarding, and it does this either outside or inside the enclave. We add connections from multiple client threads until the middlebox is saturated.

Figure 4.8 shows that the enclave did *not* have a noticeable impact on throughput, suggesting that optimizations like asynchronous system calls are not necessary for applications with I/O heavy workloads. We suspect this is due to the high interrupt rates of I/O-heavy workloads; overhead from interrupt handling overwhelms any overhead from a thread crossing the enclave boundary. Even if a developer uses asynchronous system calls, under the impression that a thread will permanently

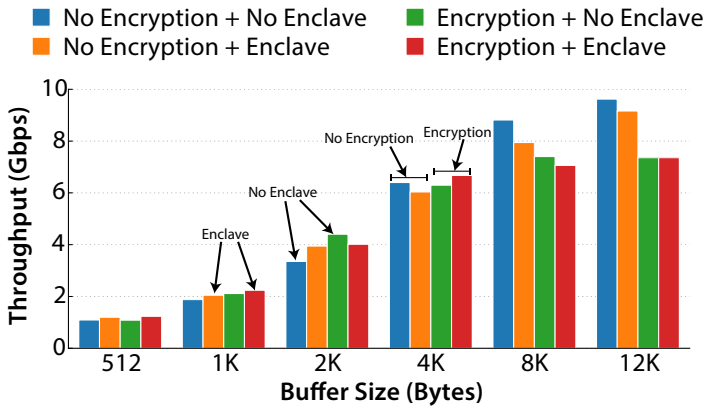


Figure 4.8: **SGX (Non-)Overhead.** Middlebox throughput with/without encryption and with/without SGX. Confidence intervals are within 1-5% of the average and differences between different scenarios for small buffers are not statistically significant.

live in the enclave, that thread will still leave the enclave whenever that core is interrupted. While pinning interrupts to a different core to avoid interrupting the enclave might help, you then pay the cost of transferring the received data from the core handling interrupts to the core running the enclave. Figure 4.8 also shows that the throughput when the middlebox decrypts/encrypts data plateaus around 7 Gbps. This is yet another source of CPU overhead that helps outweigh any performance penalty from enclave transitions.

4.6 Related Work

Middlebox as an Endpoint. In Section 4.2.2 we describe the current practice of intercepting TLS connections using custom root CA certificates. Other proposals include a 2014 IETF draft from Ericsson and AT&T [159] which would allow a proxy to intercept a TLS handshake and return a certificate identifying itself as such; if the client chooses, it can complete the handshake with the proxy and rely on the proxy to open its own TLS connection to the server. A related Cisco proposal [164] builds on this by introducing a `ProxyInfoExtension` which the proxy would use to pass the client information about the certificate and cipher suite used on the proxy-server connection. Finally, at least one ISP ships custom browsers with the certificates for its proxies built in [156]. Unfortunately, these approaches do not allow the client to authenticate the server or verify what cipher suite is used between the middlebox and the server. One could argue that, if the middlebox ran in a trusted execution environment, then it could be trusted to properly authenticate the server and use strong ciphers. However, where possible we prefer to rely on cryptography than on SGX.

Middlebox as a Middlebox. In contrast to the above approaches, which glue together separate TLS connections, several techniques have been developed for maintaining some form of end-to-end session. An IETF draft from Google has clients connect to servers directly and pass the session key to a proxy out-of-band over a separate TLS connection [187]. CloudFlare's Keyless SSL does

much the same thing for server-side delegates [218] (and [59] defines a formal security definition for keyless SSL and patches two attacks). Both of these techniques expose data to the middlebox infrastructure and fail to provide path integrity (the same session key is used on each hop).

We discuss mcTLS [177] and BlindBox [209] in Section 4.2.2.

The security concerns for network-layer middleboxes, which operate on L3 and L4 header fields (e.g., NATs and firewalls), are orthogonal to the concerns mbTLS addresses. Embark [151] is designed specifically to allow network administrators to outsource network-layer middlebox to a cloud provider without giving the cloud service private information about their network.

Network Architectures. The Delegation-Oriented Architecture (DOA) [225] provides network support for routing a packet through one or more intermediaries, but does not by itself provide all of our security properties. ICING [174] is a mechanism for enforcing path integrity and is more general than ours. Our path integrity mechanism takes advantage of the fact that mbTLS already share keys and each mbTLS record is already MAC-protected.

SGX. Protecting outsourced middleboxes with SGX was briefly discussed in [149], but without a concrete protocol or implementation. S-NFV [212] sketches a framework for implementing SGX-protected middlebox applications. PRI [202] details the design of an SGX-based IDS. Both focus on the middlebox application architecture rather than the protocol for including a middlebox in a communication session to begin with. There are also a number of TLS implementations designed to work in SGX enclaves [16, 25, 46, 230]. These libraries port unmodified TLS; we go a step further and extend the TLS handshake to include remote attestation, allowing one party to verify that the TLS session terminates inside an enclave (though this requires both to upgrade to mbTLS). Finally, there is a rapidly growing body of work on how to build SGX-protected systems (or port existing ones) [35, 44, 50, 134, 213]. These are all orthogonal to this work and could be used in concert with mbTLS to build an SGX-protected middlebox.

4.7 Conclusion

In this chapter we presented Middlebox TLS, or mbTLS, a protocol for secure multi-entity communication. Unlike previous solutions for integrating middleboxes into secure sessions, mbTLS (1) interoperates with legacy TLS endpoints and (2) can protect session data from untrusted middlebox infrastructure using trusted computing technology like Intel SGX. Our prototype implementation shows that mbTLS can indeed communicate with real, unmodified web servers and incurs reasonable overhead. Finally, we discuss the space of security properties for multi-entity communication and the trade-offs protocol designers must make among them.

Part III

Protecting Communication Metadata
(Without Giving Up Accountability)

1. How can we protect communication metadata and also hold users accountable at the same time?
2. How do we compare the privacy properties of network *architectures*?

Based on work appearing in SIGCOMM 2014 *[176] and* HotNets 2015 *[178].*

Introduction

When users communicate over the network, clearly the application data itself often contains information that ought to be private, like personal conversations, medical data, or credit card numbers. But even with the payload hidden, adversaries can still learn other sensitive information, including:

- who communicated with whom?
- who communicated at all?
- when did these communications happen?
- how many bytes were exchanged?

This "extra" information about the communication is called **metadata** and it can reveal a lot that the communicating parties might rather keep private. For example, one study found that not only are telephone numbers trivially re-identifiable, telephone metadata can reveal highly sensitive information [163]. Consider these two examples from their study:

> *Participant D placed calls to a hardware outlet, locksmiths, a hydroponics store, and a head shop in under 3 weeks.*

> *Participant E made a lengthy phone call to her sister early one morning. Then, 2 days later, she called a nearby Planned Parenthood clinic several times. Two weeks later, she placed brief additional calls to Planned Parenthood, and she placed another short call 1 month after.*

Even without the contents of these calls, it is not hard to conclude some fairly personal facts about participants *D* and *E*. Other researchers found that, using only the sizes and timestamps of encrypted Skype packets, it is possible not only to determine what language the speakers are using [234], but also to detect when they say target phrases [233]. These are not just academic concerns; communication metadata is regularly recorded and used today. Former NSA and CIA director General Michael Hayden put it succinctly: "We kill people based on metadata" [82].

III.1 Protecting Communication Metadata...

III.1.1 Properties

Once again, recall the high-level user goals presented in Section 1.1.2. The third user goal, Keep Activity Private, is the combination of two properties:

PROPERTY 4: Sender/Receiver Anonymity.
Intuitively, this means the adversary cannot tell that two particular endpoints are communicating. We formalize this idea below.

PROPERTY 5: Flow Unlinkability.
If an adversary can identify that multiple flows came from the same user, it can begin to build a history of that user's activity. (It can do this even though it may not know *who* that user is, and, in the event that future activity gives away the user's identity, it can retroactively link past activity to that user.)

Anonymity is actually many closely related ideas intertwined, and a lot of terminology has been introduced to precisely define it. We begin by reviewing this terminology from the literature and then refine it.

Anonymity in the Literature

In their 1987 paper on communication anonymity, Pfitzmann and Waidner break down anonymity into **sender anonymity**, **recipient anonymity**, and **sender-recipient unlinkability** [189]. Pfitzmann later clarified this terminology in 2001 along with Köhntopp [188] (the following definitions are taken from their paper verbatim):

> **Anonymity** is the state of being not identifiable within a set of subjects, the **anonymity set**.
>
> **Unlinkability** of two or more items (e.g., subjects, messages, events, actions, etc.) means that within this system, these items are no more and no less related than they are related given the adversary's a-priori knowledge.

Based on these fundamental definitions, they go on to define useful combinations. For example:

> **Sender anonymity** means that a particular message is not linkable to any sender and that, to a particular sender, no message is linkable.
>
> **Relationship anonymity** [sender/receiver anonymity] means that it is untraceable who communicates with whom. In other words, sender and recipient [...] are unlinkable.

Anonymity in this Document

Though intuitive, these definitions are a bit vague. Let us start with sender/receiver anonymity. What does it mean for the sender and receiver to be unlinkable? Pfitzmann and Köhntopp only go so far as to state that a sufficient condition is to have either sender anonymity or receiving anonymity—if the messages being exchanged cannot be linked to the sender (or receiver), clearly the sender (receiver) cannot be linked to the receiver (sender). We argue that, while true, this

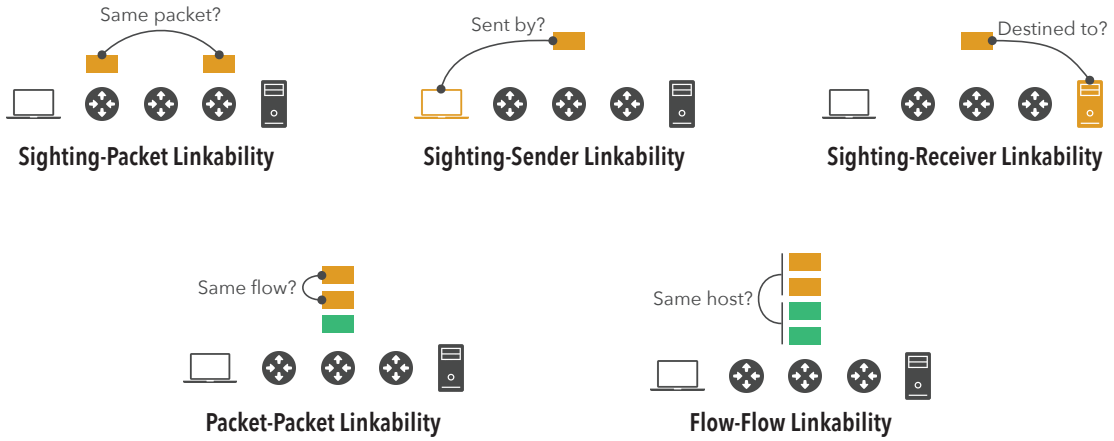


Figure III.1: **Linkability.** Illustrations of each type of linkability. Each rectangle represents a packet sighting at the router below it.

characterization is not particularly useful for two reasons. First, if the adversary has a vantage point on the first (last) link (a common scenario), it knows who the sender (receiver) is based on network topology regardless what information is in the packet, so in this case it is impossible to have sender (receiver) anonymity.

Second, the definition above for sender anonymity is not a very useful one. Consider the following example: a sender, S, sends a packet, which at some point traverses a NAT. Before the NAT, the packet's source address is that of S; after the NAT, its source address is that of S's home network. To anyone observing the packet before the NAT, the sender is clearly S. To anyone observing the packet after the NAT, the sender could be any host in S's home network. Is the packet linkable to S or not?

This example illustrates the futility of talking about packet-host linkability. The problem is one of granularity: talking about linkability at the granularity of packets is not useful, because packets can be changed as they move through the network, which can in turn change whether they can be linked to the sender or receiver. What we really care about is the interplay between three things: **sighting-host linkability**, **sighting-packet linkability**, and **packet-flow linkability**. First we offer precise definitions for these terms and then we compose them to offer our own definition of sender/receiver anonymity.

Linkability. Pfitzmann and Köhntopp's definition refers to linkability between two or more "items." There are a few different types of item-item pairs we care about (see Figure III.1). The first four are "fundamental" in the sense that we cannot a priori define *how* the adversary determines if a pair of items are linkable; adversaries of varying power might come to different conclusions.

Sighting-Packet Linkability. Are these sightings the same packet? The adversary can determine this using any means available to it (e.g., by comparing payloads or timing analysis). We denote this by $\mathsf{LinkableSightings}_P(\cdot,\cdot)$.

Sighting-Flow Linkability. Are these sightings part of the same flow? The adversary can determine this using any means available to it (e.g., timing analysis or TCP sequence numbers).

We denote this by $\mathsf{LinkableSightings}_F(\cdot,\cdot)$.

Sighting-Sender Linkability. Was the packet observed in this sighting sent by host H? The adversary can determine this using any means available to it (e.g., from the packet's source address or by sniffing the link connecting H to the network). We denote this by $\mathsf{LinkableSightingSender}_H(\cdot)$.

Sighting-Receiver Linkability. Is the packet observed in this sighting destined for host H? As above, the adversary can determine this in many ways. We denote this by $\mathsf{LinkableSightingReceiver}_H(\cdot)$.

The last two are "composite" definitions, in the sense that they build on the above definitions and do not require (additional) assumptions about the adversary.

Packet-Packet Linkability. Are these packets part of the same flow? When we talk about packet linkability, a packet is defined as *a set of linkable sightings*. We denote this by $\mathsf{LinkablePackets}(\cdot,\cdot)$ and define it as follows:

$$\mathsf{LinkablePackets}(P_1,P_2) := \exists a \in P_1, b \in P_2 \;\; s.t. \;\; \mathsf{LinkableSightings}_F(a,b) \tag{4.1}$$

Flow-Flow Linkability. Are these flows from/to the same host H? When we talk about flow linkability, a flow is defined as *a set of linkable packets*. We denote this by $\mathsf{LinkableFlows}_H(\cdot,\cdot)$ and define it as follows:

$$\begin{aligned}\mathsf{LinkableFlows}_H(F_1,F_2) := \exists a \in F_1, b \in F_2 \;\; s.t.& \\ [\mathsf{LinkableSightingSender}_H(a) \wedge \mathsf{LinkableSightingSender}_H(b)]& \\ \vee\, [\mathsf{LinkableSightingReceiver}_H(a) \wedge \mathsf{LinkableSightingReceiver}_H(b)]&\end{aligned} \tag{4.2}$$

Note: in these definitions, we use the terms "packet" and "flow" from the point of view of the adversary. For example, a packet entering and leaving a proxy that re-encrypts the payload might appear to some adversaries to be two different packets even though, in a ground truth sense, they are the same packet.

Sender/Receiver Anonymity. We are now able to define sender/receiver linkability, which we use as our definition of sender/receiver anonymity. Intuitively, a sender and receiver are linkable if we have (1) a sighting that is linkable to the sender, (2) a sighting that is linkable to the receiver, and (3) those two sightings are themselves linkable. Sender/receiver linkability is formalized as follows:

$$\begin{aligned}&\mathsf{LinkableSenderReceiver}(H_1,H_2) := \exists a, b \;\; s.t. \\ &\mathsf{LinkableSightingSender}_{H_1}(a) \wedge \mathsf{LinkableSightingReceiver}_{H_2}(b) \wedge \mathsf{LinkableSightings}_F(a,b)\end{aligned} \tag{4.3}$$

Figure III.2 puts these new definitions in context in the concept map from Figure 1.1. This will be useful below when we discuss techniques for providing these properties.

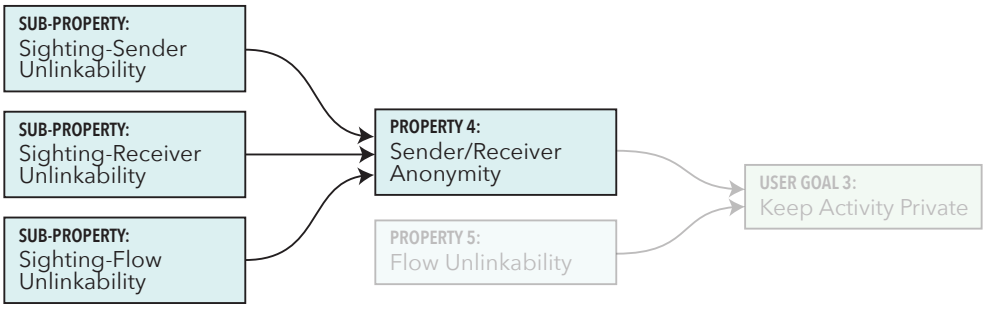


Figure III.2: **Sender/Receiver Anonymity.** A breakdown of the properties contributing to sender/receiver anonymity.

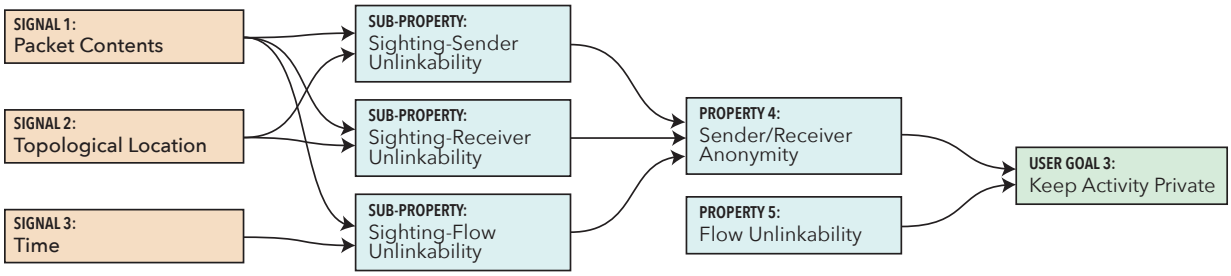


Figure III.3: **Signals.** A network adversary uses three types of *signals* to break users' anonymity: packet contents, topological location, and timing.

III.1.2 Techniques

Protecting communication metadata—that is, providing anonymity—is less straightforward than protecting the data itself. Before we discuss methods for doing so, first consider the problem from the adversary's point of view. How does an adversary learn about users' communication patterns by observing network traffic? Every time the adversary sees a packet at one of its vantage points (a **sighting**), it learns from three primary **signals**:

SIGNAL 1: Packet Contents
The bits in a packet explicitly encode information that may be useful to an adversary. For example, if two hosts with public IP addresses are communicating without any intervening proxies, the source and destination IP addresses directly link the packet to the sender and receiver. And the standard "5-tuple" (network addresses, transport port numbers, and transport protocol number) is an easy way to group packets into flows. We assume application payloads are encrypted and therefore consider leakage of identity information in payloads out of scope (though the adversary may use equality tests on encrypted payloads to link multiple sightings of the same packet).

SIGNAL 2: Topological Location
A vantage point's topological location restricts the sets of possible senders or receivers.

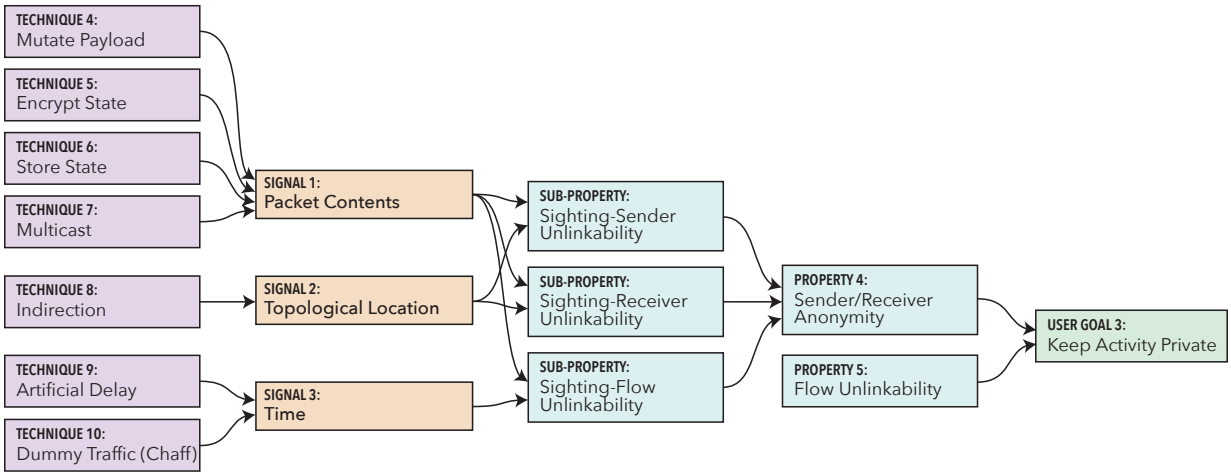


Figure III.4: **Metadata Protection Techniques.** Techniques for hiding communication metadata leaked by the signals.

This is particularly true on the first and last hops, where the presence of a packet uniquely identifies the sender or receiver.

SIGNAL 3: Time
Packet timing (and ordering) can help the adversary link sightings of the same packet at different vantage points even if the contents have changed (e.g., before and after a Tor relay). Timing can also help link packets in the same flow—packets in a burst are more likely to be part of the same flow.

To prevent adversaries from learning anything useful from this information, a few categories of defenses have emerged (Figure III.4). The first four deal with hiding information typically leaked by Packet Contents. Let us start with the payload:

TECHNIQUE 4: Mutate Payload
We assume the payload is encrypted, which means its only use to the adversary is linking sightings of the same packet. This can be prevented by *mutating the payload* (e.g., re-encrypting it under a new key). Tor relays do this.

Hiding headers is trickier, because certain pieces of sensitive information are needed by certain entities to carry out network communication, namely (1) the destination (needed by routers to deliver the packet), (2) the source (needed by return path routers to deliver the reply), and (3) a flow ID (needed by routers and some middleboxes for traffic engineering and by endpoints for demultiplexing). (Note that "destination" and "source" do not necessarily mean addresses in a header, but rather the knowledge, in any form or at any granularity, of where the packet should be delivered and where replies should be sent.) This **communication state** is needed for successful communication, but that does not mean it needs to be publicly visible. We make three observations,

each leading to a general technique for hiding communication state:

TECHNIQUE 5: Encrypt State
Observation 1: Not all devices need all of the information.
For example, the receiver does not need to know the source so long as the network is able to deliver the reply (typically routers know how to deliver responses because the sender passes the receiver its address, which the receiver uses as the destination address in its response). Furthermore, each device may not need to know it at its fullest granularity (e.g., the first hop router does not need to know the exact destination, it only needs a rough idea of which direction to send the packet). This suggests forwarding state could be broken into pieces which are individually *encrypted* for the routers that need them.

TECHNIQUE 6: Store State
Observation 2: Communication state does not need to be carried in each packet.
It can also be *stored* on the devices that need it rather than being carried around the network, where it is visible at more vantage points. For example, a **network address translator (NAT)** sits at the border of two networks, an "internal" network and an "external" network, and rewrites the addresses of the internal hosts in all packets leaving the internal network to a single, shared address. It also assigns the flow a new TCP or UDP port number, which it uses to disambiguate internal hosts when inbound traffic arrives; this *internal address* ↔ *external port* mapping is stored on the NAT. By doing this, sightings of a packet outside the NAT cannot be attributed to a specific sender.

Another example can be found in a future Internet architecture called Named Data Networking (NDN) [138]. Packets in NDN do not contain source addresses at all; instead, packets leave "breadcrumbs" in the routers they traverse and response traffic follows these breadcrumbs back to the initial sender.

TECHNIQUE 7: Multicast
Observation 3: Communication may still be possible even with less than complete information.
For example, even without a destination address, routers could *multicast* a packet to many hosts in the network. The intended recipient looks for a secret (e.g., encrypted) signal that the packet is for it.

The next technique is used to hide the Topology signal:

TECHNIQUE 8: Indirection
Forwarding traffic through some kind of relay—like an anonymizing proxy, a VPN endpoint, or a circuit of Tor relays—takes it off the default routing path from sender to receiver. This limits what an adversary can learn from the topological location of a vantage point, since the packet's presence does not necessarily mean that the true destination is nearby.

And, finally, the remaining two techniques are for hiding Timing information:

TECHNIQUE 9: Artificial Delay
An intermediary in the network can delay packets artificially to hide correlations in packet arrival timings. This is often done in batches—the intermediary delays packets until it has a full batch, then releases them in a random order. (Such an intermediary is called a **mix**.)

TECHNIQUE 10: Dummy Traffic (Chaff)
Rather than introducing artificial delay to hide timing patterns, "dummy" packets can be used to fill gaps between real packets so that the adversary always sees traffic flowing at a fixed rate. This practice is sometimes called **chaffing** and the dummy traffic is **chaff**.

As a concrete example, consider Tor [96]. Tor is a real-world anonymous communication service that combines techniques 4, 5, and 8 in a primitive called **onion routing** [118]. Tor clients forward packets through a series of proxies called **relays** (indirection), each of which re-encrypts the payload to prevent adversaries from linking sightings before and after (payload mutation). In Section III.3.1, we will discuss other concrete instantiations of these techniques from research and practice.

III.2 ...Without Giving Up Accountability

Researchers have long recognized the need for anonymous communication systems to protect metadata, and we will discuss several concrete instantiations of the techniques introduced above in Section III.3.1. Furthermore, these systems are not merely academic endeavors; several have been deployed and have sizeable user bases, and the recent attention drawn to mass surveillance by intelligence agencies has increased user awareness of existing services like VPNs and Tor [23, 66, 68, 128, 182]. Section III.3.1 describes concrete systems in detail.

However, using systems like Tor to hide senders' identities, while good from a privacy perspective, also allows misbehavior to go unpunished. Network operators and law enforcement use packet source addresses to track down users who abuse the network by doing things like downloading illegal content, sending SPAM, or harassing other users. If users can hide their true source addresses at will, these authorities lose the ability to identify miscreants, resulting in a loss of **accountability**.

At the other extreme, mechanisms that improve accountability often do so at the expense of privacy. For example, the Accountable Internet Protocol (AIP) [40] prevents source address spoofing by cryptographically linking packets to their senders.

One way to think about this, illustrated by Figure III.5, is the following. The current Internet architecture provides virtually no support for privacy *or* accountability. Existing techniques, like Tor and AIP, provide one at the expense of the other. Our goal is to provide a practical balance of both.

We argue that providing this balance requires support from the network. Clearly if we want accountability-related features like source address spoofing prevention or the ability to block attack traffic, we need help from the routers; once the network delivers spoofed or otherwise malicious

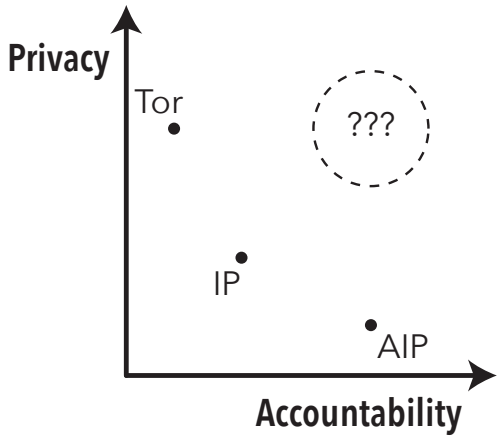


Figure III.5: **Accountability vs. Privacy.** IP provides little support for privacy or accountability. Existing techniques provide one at the expense of the other. Our goal is to achieve a useful balance of both.

packets to a victim, the damage may already be done even if the victim can identify the packet as malicious and drop it. At the same time, anonymity systems are designed to break the link between the source address visible in the packet and the true sender. This breaks the traditional assumption that the source address is a "handle" we can use to identify misbehaving senders. The network needs to acknowledge this and provide a suitable replacement.

In an effort to address this tussle between privacy and accountability, the chapters in this part make the following contributions:

- **Architectural support for balancing privacy and accountability.** In Chapter 5 we introduce a new network architecture, the Accountable and Private Internet Protocol (APIP), that explicitly balances privacy and accountability—and, in fact, offers more of each than the current Internet, without introducing significant overhead in the default case.
- **Techniques for quantifying privacy.** While—and after—we designed APIP, we found we struggled to rigorously compare it to other architectures in terms of privacy. This turned out to be a tricky, open problem; Chapter 6 describes our efforts on how to quantify "how much privacy" a network architecture provides.

III.3 Related Work

III.3.1 Anonymity Systems

In this section, we summarize existing anonymous communication systems from research and practice, binning them loosely into categories based on the strength of their guarantees and the overhead they impose. (Danezis et al. provide a more thorough overview of the field through 2009 [88].) These systems are generally concrete instantiations of one or more of the high-level techniques presented in Section III.1.2. A general rule of thumb for understanding their design is that any anonymity system is a balancing act between three desirable things: (1) low latency, (2) high bandwidth, and (3) strong anonymity.

Lightweight. The first class of systems are those that attempt to provide the best anonymity possible without inflating latency or reducing bandwidth. Though unable to stand up to powerful, global adversaries like a malicious ISP or a government spy agency, this class of solutions is well-suited for more limited attackers like local Wi-Fi sniffers or end servers.

Anonymizing Relay. The simplest privacy service is a single relay, either at the network layer (a VPN) or at the transport or applications layers (a proxy). Examples include Anonymizer [4], VyprVPN [29], ExpressVPN [12], Hide My Ass! [15], and StrongVPN [22]. These services work by encrypting the sender's packets (Encrypt State) and tunneling them to a proxy that decrypts them (Mutate Payload) and forwards them to the ultimate destination. The true destination's address is not visible before the relay; after the relay, the true sender's address is unavailable Furthermore, the detour through the relay can help to hide information revealed by topology (Indirection).

Network-Layer Onion Routing. This bin includes LAP [131], Dovetail [201], HORNET [77], and an unnamed architecture designed around the principle of least privilege [157, 158]. At a high level, these approaches all work by (1) sending each packet along the path it would have taken anyway, without the privacy protections, to avoid path stretch and (2) exposing, at each router, only the forwarding state needed to pick the next hop (e.g., by encrypting each router's next-hop information).

Medium-weight.

Onion Routing. A single relay is sufficient against an adversary with limited capabilities; if an adversary is more powerful (e.g., has multiple vantage points or observations collected over time), this idea can be extended by encrypting a packet multiple times and forwarding it through a series of relays, each of which strips off a layer of encryption and forwards the packet to the next hop. This process is known as **onion routing** [118]. The most well-known instantiation of this idea is Tor [96], a popular onion routing service with over two million daily users [28]. However, since Tor attempts to introduce as little delay as possible, it is still susceptible to timing attacks—an adversary can link packets entering and leaving a Tor relay even though the payload has been re-encrypted. Like anonymizing proxies, onion routing uses a combination of Encrypt State, Mutate Payload, and Indirection (it just uses more of each).

Onion routing traditionally relies on a PKI (doing the onion encryption requires knowing the public keys of the relays). An interesting alternative approach takes advantage of path diversity in the network and uses *information slicing* to provide similar guarantees without a PKI [145].

Heavyweight. To resist even traffic analysis attacks by powerful, global adversaries, a variety of heavyweight techniques can be used. In exchange for stronger anonymity guarantees, each one sacrifices some combination of low latency and high bandwidth.

Mix Nets. Building on onion routing, imagine now that each relay gathers a batch of packets from multiple users of the system, shuffles them [110, 179], and releases them. Such a relay is called a **mix** [75]. Examples based on a fixed infrastructure of mixes include Babel [122], Mixmaster [169], Mixminion [90], and the Java Anonymous Proxy [54]; Drac [89], Blindspot [112], and $\mathcal{P}^5$ [210] use a peer-to-peer model. The trouble is, unless the mix net's users generate a very large volume of traffic, the system has to either (1) wait for significant periods of time to gather a big enough batch to shuffle and release (Artificial Delay), adding latency overhead, or (2) introduce Dummy Traffic

(Chaff) to "fill in the gaps," adding bandwidth overhead. More recent systems, like Aqua [154] and Herd [153], try to reduce these overheads by carefully balancing chaff vs. delay for the anticipated workload (Aqua targets bulk file transfer, like BitTorrent, and Herd targets VoIP) and by playing other tricks.

DC-Nets. Another approach, also due to Chaum, is a cryptographic construction called a **Dining Cryptographers Network**, or DC-Net [76, 119]. DC-nets are peer-to-peer systems which divide communication into *rounds*; in each round, one peer can publish a public, anonymous message. Each pair of peers shares a symmetric key; during a round, the peer whose turn it is to speak uses these keys to generate pseudo-random strings for each other peer. It then XORs these strings together along with its message and publishes the result; everyone else can recover the message by generating the same pseudo-random strings and XORing them with the published ciphertext. The trick is making this scale: all participants in the DC-net need pairwise keys with one another, and each round requires n^2 communication. Dissent [86, 231] introduces a hybrid infrastructure/P2P model that allows their system to scale to more users. Another problem with DC-nets is their susceptibility to insider "jamming" attacks in which a malicious peer speaks out of turn, preventing honest users from speaking. Herbivore [117] combats this by breaking the network into smaller "anonymity cliques" while Verdict [87] uses cryptographic mechanisms to prevent jamming.

Mailbox Systems. Finally, there is yet another class of anonymous communication systems that are essentially databases where a sender/receiver write/read messages to/from a pre-arranged database entry. The systems are designed to allow these writes and reads to happen without anyone learning which key is being written or read. Three such systems are P^3 [150], Vuvuzela [223], and Pung [41]. Vuvuzela's guarantees are based on differential privacy. Riposte [85] uses these ideas to implement a public message board system (think: anonymous Twitter) rather than supporting two-way conversations.

III.3.2 Measuring Anonymity

There are a number of proposals for measuring network layer privacy, many of which fall into two broad classes.

The first is a group of techniques for characterizing how much the adversary knows about the anonymity set, beginning with simply the size of the set [55, 76]. However, unless the distribution of suspicion across hosts in the anonymity set is uniform, set size does not mean much; for non-uniform distributions, the maximum probability [221] or the entropy of the distribution [92, 206] are more meaningful. (Reiter and Rubin capture these ideas with an informal scale with intuitive levels like "beyond suspicion," "probable innocence," and "possible innocence" [193].) Entropy captures *how much the adversary knows* about who is communicating, not *how much it learned* from observing the system; mutual information can be used to characterize how much the communication protocol is to blame for what the adversary knows [74, 91]. The authors of [93] propose a Bayesian definition of anonymity that can reason about what an attacker learns by combining multiple sources of knowledge, some of which may be incorrect. Entropy and mutual information both describe the adversary's ability to link *one particular* output to its corresponding input, which can be misleadingly optimistic. A better measure considers the likelihood of all possible matchings of inputs to outputs [101, 116].

The second group of techniques is formal methods, which specify an anonymity system in some formal language and then automatically verify properties about it. Some use process calculi [57, 203], which is good at encoding protocols but less so at expressing privacy properties. Others use epistemic logic [73, 111, 219], which clearly captures privacy properties but makes it more difficult to encode the protocol. Function views can combine the benefits of a process calculus with epistemic logic [133].

Chapter 5 Balancing Privacy and Accountability in the Network

Though most would agree that accountability and privacy are both valuable, today's Internet provides little support for either. Previous efforts have explored ways to offer stronger guarantees for one of the two, typically at the expense of the other; indeed, at first glance accountability and privacy appear mutually exclusive. At the center of the tussle is the source address: in an accountable Internet, source addresses undeniably link packets and senders so hosts can be punished for bad behavior. In a privacy-preserving Internet, source addresses are hidden as much as possible.

In this chapter, we argue that a balance *is* possible. We introduce the Accountable and Private Internet Protocol (APIP), which splits source addresses into two separate fields—an *accountability address* and a *return address*—and introduces independent mechanisms for managing each. Accountability addresses, rather than pointing to hosts, point to *accountability delegates*, which agree to vouch for packets on their clients' behalves, taking appropriate action when misbehavior is reported. With accountability handled by delegates, senders are now free to mask their return addresses; we discuss a few techniques for doing so.

5.1 Introduction

Today's Internet is caught in a tussle [80] between service providers, who want accountability, and users, who want anonymity (Keep Activity Private). Each side has legitimate arguments: if senders cannot be held accountable for their traffic (e.g., source addresses are spoofable), stopping in-progress attacks and preventing future ones becomes next to impossible. On the other hand, there are legitimate anonymous uses of the Internet, such as accessing medical web sites without revealing personal medical conditions, posting to whistleblowing web sites, or speaking out against an oppressive political regime.

At the network layer, mechanisms for providing one or the other often boil down to either strengthening or weakening *source addresses*. In an accountable Internet, source addresses undeniably link packets and senders so miscreants can be punished for bad behavior, so techniques like egress filtering and unicast reverse path forwarding (uRPF) checks aim to prevent spoofing. In a private Internet, senders hide source addresses as much as possible, so services like Tor work by masking the sender's true source address.

We argue that striking a balance between accountability and privacy is fundamentally difficult because the IP source address is used both to identify the sender (accountability) *and* as a return

address (privacy). In fact, the function of the source address has evolved to be even more complex, serving a total of five distinct roles: packet sender, return address, error reporting (e.g., for ICMP), accountability (e.g., uRPF), and to calculate a flow ID (e.g., as part of the standard 5-tuple).

This chapter explores how changes to the network architecture could help by asking the question, "What could we do if the accountability and return address roles were separated?" Our answer, the Accountable and Private Internet Protocol (APIP), does just that, creating an opportunity for a more flexible approach to balancing accountability and privacy in the network. APIP utilizes the accountability address in a privacy-preserving way by introducing the notion of *delegated accountability*, in which a trusted third party vouches for packets and fields complaints. With accountability handled by delegates, senders have more freedom to hide return addresses. We make the following contributions:

- An analysis of the roles of the source address in today's Internet.
- The definition of design options for an accountability address and the accompanying mechanisms for holding hosts accountable in a privacy-preserving way.
- An analysis of the impact of these design options on the privacy-accountability tradeoff.
- The definition and evaluation of two end-to-end instantiations of APIP.

The remainder of the chapter is organized as follows. After teasing apart the various roles of the source address (Section 5.2), Section 5.3 discusses challenges in balancing accountability and privacy. Section 5.4 gives a high-level overview of APIP. Section 5.5 describes possible designs for delegated accountability while Section 5.6 analyzes their implications for privacy. Section 5.7 discusses real-world deployment issues and presents two example end-to-end instantiations of APIP. We evaluate the feasibility of APIP in Section 5.8 and finish with a discussion of related work (Section 5.9) and conclusion (Section 5.10).

5.2 Source Address Overload

We now investigate the roles of source addresses, since they play a key role in the seemingly fundamental conflict between accountability and privacy in the network. Source addresses today attempt to fulfill at least five distinct roles:

1. **Return Address.** This is a source address's most obvious role: the receiving application uses the source address as the destination for responses. (This is, for example, built into TCP connection establishment.)
2. **Sender Identity.** Historically, source addresses have been used as a crude (and ineffective) way of authenticating a sender or to link multiple sessions to a single "user."
3. **Error Reporting.** If a packet encounters a problem, the ICMP error message is directed to the source address.
4. **Flow ID.** Source addresses are one component of the 5-tuple used to classify packets into flows, both in the network (monitoring, traffic engineering) and at endpoints (demultiplexing).
5. **Accountability.** Techniques such as uRPF checks and egress filtering can be viewed as weak accountability mechanisms protecting against certain types of address spoofing. Recent work

Role	Where Used	Layer	Comments
Return Address	Destination	Transport	Routers forward purely based on the destination address; the return address is used only by the destination.
Sender Identity	Destination	Application	No longer used to authenticate users, but may be used to, e.g., track "users" across sessions in web access logs.
Error Reporting	Routers	Network	Destination for error messages.
	Destination	Network	
Flow ID	Destination	Transport	End-hosts need a way to demultiplex flows.
	Routers	Network	Routers distinguish flows for traffic monitoring/engineering.
Accountability	Routers	Network	In designs like AIP, routers may require a valid (challengeable) source address.
	Destination	Network	It must be possible to identify and shut down a malicious flow.

Table 5.1: **Source Address Roles.** The roles a source address plays and where each is used.

offers stronger protection than that offered by IP. For example, AIP [40] uses cryptographic identifiers as source addresses that can be used to verify that the host identified really did send the packet.

Somewhat to our surprise, many proposed architectures use source addresses for the same purposes. This includes proposals that are very different from IP, such as architectures that use paths or capabilities, rather than addresses, to identify a destination. For instance, SCION [240] headers include AS-level paths selected jointly by the ISPs and source and destination networks to specify how to reach the destination. However, each packet still has an AIP-style source identifier that fulfills the above roles. Also, in ICING [174] and capability-based architectures such as SIFF [236], TVA [237], or FANFARE [211], packets carry pre-approved router-level paths, but they also carry traditional source and destination addresses.

To understand the impact of repurposing the source address as an accountability address in APIP, we ask two questions about each role:

(1) Is it needed by the network? If not, it can be moved deeper in the packet, opening up more design options and simplifying the network header.

(2) Is it needed in every packet? If not, it could be stored elsewhere, e.g., on the routers or end-hosts that will use it. This simplifies the packet header, but may add complexity to protocols that have to maintain the state.

Table 5.1 summarizes the answers to these questions. Two high-level takeaways emerge: (1) not all roles involve the network, and (2) some information is not needed in every packet. The following observations are particularly relevant to the accountability versus privacy tussle:

1. **The accountability role is the network's primary use of source addresses.** Error reporting benefits the host, not the network. Hosts could choose to forgo error reports for the sake of privacy or have them sent to the accountability address. Flow ID calculation could use the accountability address.
2. **The return address role is not used by the network at all.** It could be moved deeper in the packet, encrypted end-to-end, and/or omitted after the first packet of a flow.

5.3 Accountability versus Privacy

A number of research efforts focus on improving either accountability or sender privacy in the network, but unfortunately this often comes at the price of weakening the other. To illustrate this point, we summarize one well-known technique for each and then elaborate on the goals of this chapter.

5.3.1 Previous Work

Accountability and Nothing But. The Accountable Internet Protocol (AIP) [40] is a network architecture whose primary objective is accountability. Each host's *endpoint identifier* (EID) is the cryptographic hash of its public key, and AIP introduces two mechanisms that use these "self-certifying" EIDs to hold hosts accountable.

First, first-hop routers in AIP prevent spoofing by periodically "challenging" a host by returning the hash of a packet it purportedly sent. Hosts maintain a cache of hashes of recently sent packets and respond affirmatively if they find the specified packet hash; the response is signed with the private key corresponding to the source EID. If a challenge elicits no response or the response has an invalid signature, the router drops the original packet. Second, AIP proposes a *shutoff* protocol: a victim under attack sends the attacking host a *shutoff packet*, prompting the attacker's NIC to install a filter blocking the offending flow. Shutoff packets contain the hash of an attack packet (to prove the host really sent something to the victim) and are signed with the victim's private key (to prove the shutoff came from the victim).

AIP suffers from three important limitations: first, cryptographically linking senders to their packets' source addresses precludes any possibility of privacy. Second, though bad behavior is always linkable to the misbehaving host, AIP does not facilitate a long-term fix—the shutoff protocol is only a stop-gap measure. Finally, AIP requires that well-intentioned owners install "smart NICs" that implement the challenge and shutoff protocols, since a compromised OS could ignore shutoffs. We draw heavily on ideas from AIP while addressing these limitations.

Privacy and Nothing But. The best available solution for hiding a return address is using a mix net or onion routing service like Tor [75, 192, 193]. Observers in the network only see the identity of the two onion routers on that link in the Tor path. Of course, accountability is much more difficult to achieve since the identity of the sender is hidden inside the packet, behind one or more layers of encryption. Liu et al. propose an architecture that offers a high degree of privacy by baking Tor into the network itself [157]. However, in addition to the lack of accountability, the increased header overhead and latency make Tor unsuitable as a default, "always-on" solution.

5.3.2 Goals and Threat Model

The examples show that the source address represents a control point in the tussle between privacy and accountability. Unfortunately, it is a very crude one since there seem to be only two settings: privacy (x)or accountability. The high level goal of this chapter is to redefine the source address so it can properly balance the accountability and privacy concerns of providers and users.

Accountability. At the network layer, by accountability we mean that *hosts cannot send traffic with impunity: malicious behavior can be stopped and perpetrators can be punished*. Specifically, we would like our design to have the following three properties:

1. Anyone can verify that a packet is "vouched for"—a known entity is willing to take responsibility if the packet is malicious.
2. Malicious flows can be stopped quickly.
3. Future misbehavior from malicious hosts can be prevented (i.e., by administrative or legal action).

Privacy. Our goal is to give users more anonymity than they have in the current Internet. At the same time, since changes to the network layer affect all traffic in the Internet, the mechanisms we introduce should be lightweight enough that they do not impose overhead in the default case. For this reason, we target only the Packet Contents signal; hiding the Topological Location and Time signals require introducing significant overheads (Section III.1.2).

Therefore, APIP *allows senders to hide their network addresses* from some combination of third-party observers in the source domain, transit ISPs, and the destination host. Our threat model excludes the following:

- We do not consider anonymity from the operator of the source domain itself (since it can identify the sender based on the physical "port" through which the packet entered the network).
- APIP does not hide a packet's destination; senders wishing to make their packets unlinkable to the destination should use solutions such as Tor.
- Timing attacks—e.g., using traffic analysis to link packets arriving at a host with response packets leaving the host—are out of scope.
- Though we do not introduce new techniques for flow anonymity—i.e., the inability of observers to link packets belonging to the same flow, which can be useful if features like the size of a communication are sensitive—we discuss how APIP affects the linkability of packets in a flow.
- Application-layer privacy concerns are out of scope.

Finally, note that APIP is not limited to any one particular address hiding mechanism, but rather makes it possible for senders to hide their addresses any way they see fit.

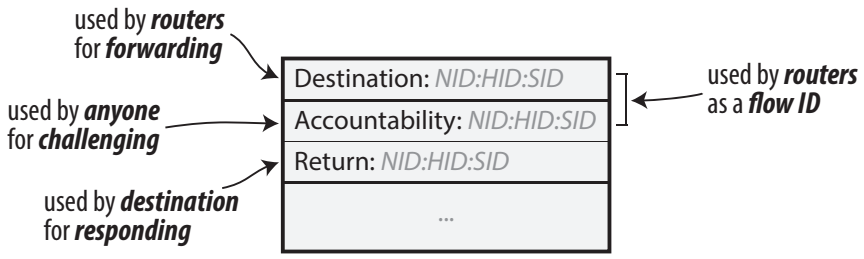


Figure 5.1: **APIP Headers.** Packet carry a destination address (used by routers for forwarding), an accountability address (used to report malicious packets), and an optional return address (used by the receiving endpoint for responding).

5.4 Basic Design

The Accountable and Private Internet Protocol (APIP) ***separates accountability and return addresses***. A dedicated accountability address allows us to address the limitations of an accountability-above-all-else approach like AIP by introducing ***delegated accountability***. Rather than identifying the sender, a packet's accountability address identifies an *accountability delegate*, a party who is willing to vouch for the packet. With accountability handled by delegates, senders are free to ***mask return addresses*** (e.g., by encrypting them end-to-end or using network address translation) without weakening accountability.

Addressing. We think APIP is applicable to many different network architectures, so as much as possible we avoid making protocol-specific assumptions. To discuss source addresses generally, we adopt three conventions.

First, each packet carries at least two addresses (Figure 5.1): (1) a *destination address* (used to forward the packet) and (2) an *accountability address* (identifying a party—not necessarily the sender—agreeing to take responsibility for the packet). It may also carry a *return address* (denoting where response traffic should be sent) as a separate field in the packet. Return addresses may not be present in all packets, e.g., they may be stored with connection state on the receiver. Also, as we discuss later, the return address may not always be part of the *network* header.

Second, an address consists of three logical pieces: (1) a *network ID* (NID), used to forward packets to the destination domain, (2) a *host ID* (HID), used within the destination domain to forward packets to the destination host, and (3) a *socket ID* (SID), used at the destination host to demultiplex packets to sockets. We write a complete address as *NID:HID:SID*. These logical IDs may be separate header fields or could be combined (e.g., an IP address encodes both an NID and an HID; the port number serves as an SID).

Finally, to simplify our description of APIP, we initially assume that HIDs are *self-certifying*, as defined by AIP, to bootstrap trust in interactions with accountability delegates. We relax this assumption in Section 5.7.3.

Life of a Packet. Figure 5.2 traces the life of a packet through APIP.

❶ The **sender** sends a packet with an *accountability address* identifying its **accountability delegate**. If a *return address* is needed, it can be encrypted or otherwise masked.

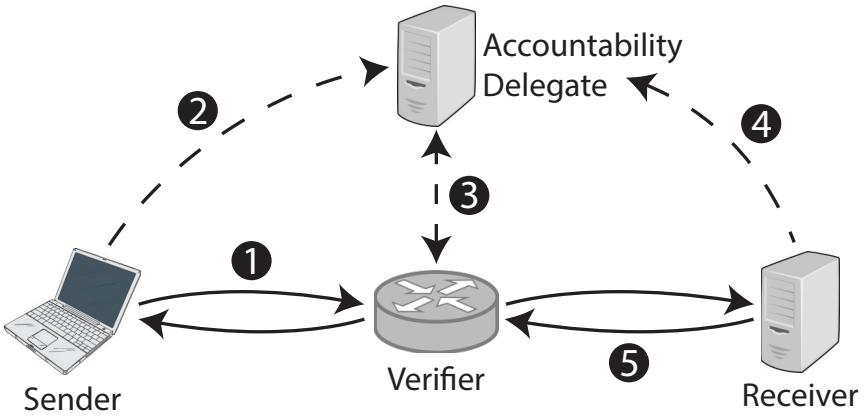


Figure 5.2: **APIP Overview.** The life of a packet in APIP.

❷ The **sender** "briefs" its **accountability delegate** about the packet it just sent.

❸ A **verifier** (any on-path router or the receiver) can confirm with the **accountability delegate** that the packet is a valid packet from one of the delegate's clients. Packets that are not vouched for are dropped.

❹ If the **receiver** determines that packets are part of a malicious flow, it uses the *accountability address* to report the flow to the **accountability delegate**, which stops verifying (effectively blocking) the flow and can pursue a longer term administrative or legal solution.

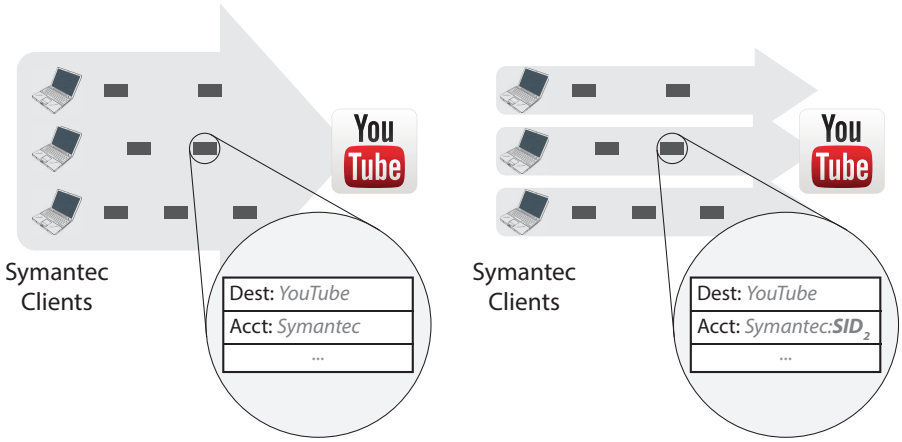
❺ The **receiver** uses the *return address* in the request as the *destination address* in the response.

It is useful to identify the key differences between APIP and the AIP and Tor protocols discussed in Section 5.3. Delegated accountability offers two key benefits over AIP. First, it dramatically improves sender privacy: only the accountability delegate, not the whole world, knows who sent a packet. Second, it offers a more reliable way of dealing with malicious flows compared to a smart NIC. Third, it offers a clearer path to long-term resolution to bad behavior. For example, the delegate can contact the well-intentioned owner of a misbehaving host out-of-band (e.g., requiring them to run anti-virus tools). While Tor provides stronger privacy properties than APIP, by simply changing how source addresses are treated, APIP can provide sender privacy with much lower overhead since the return address can be hidden from the network. Techniques for doing so (Section 5.6) are lightweight enough to be viable options for "default on" use.

5.5 Delegating Accountability

This section describes how accountability can be delegated. We will assume delegates can be trusted, e.g., their role is played by a reputable commercial company or source domain. We discuss the problem of rogue delegates in Section 5.7.1. APIP defines four aspects of delegate operation: the form of the address used to reach a delegate plus the three operations all delegates must support — the delegate "interface," so to speak. Delegates expose one operation to their clients:

brief(packet, clientID): Whenever a host sends a packet, it must "brief" its delegate—if the delegate is to vouch for the packet on behalf of the sender, it needs to know which packets its clients actually sent.



(a) Without *SID*s. (b) Per-host *SID*s.

Figure 5.3: **Flow Granularity.** Adding *SID*s to accountability addresses for flow differentiation.

To the outside world, accountability delegates offer two operations, borrowed largely from AIP:

verify(packet): Anyone in the network can challenge a packet; its accountability delegate responds affirmatively if the packet was sent by a valid client and the flow has not been reported as malicious.

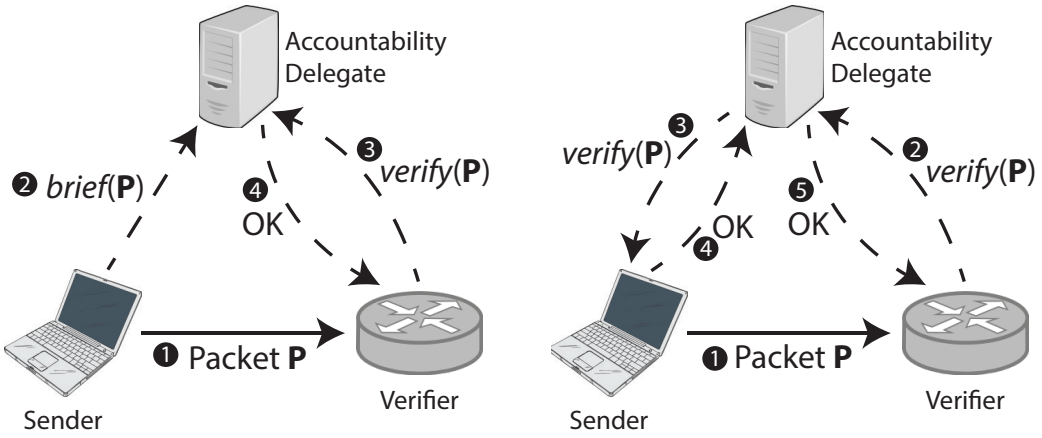
shutoff(packet): Given an attack packet, the victim can report the packet to the accountability delegate; in response, the delegate stops verifying (blocks) the flow in question and pursues a long term solution with the sender.

We now discuss options for constructing the accountability address and for implementing the delegate interface.

5.5.1 Accountability Addresses

Accountability addresses serve two related functions. First, the address is used to send verification requests and shutoffs to an accountability delegate. The NID:HID portion of the address is used to direct messages to the delegate server. Second, routers often need to identify flows, e.g., for traffic engineering (TE) or monitoring purposes, and today source addresses are often part of the flow ID. The granularity of this ID is even more important in APIP since traffic is verified (and blocked) per flow. In this section, we discuss the implications of replacing source addresses with accountability addresses for flow identification.

Creating Flow IDs Routers construct flow IDs using information available in the network and transport headers. However, in APIP, if an accountability address merely points to a delegate, packets from all clients of a particular delegate will be indistinguishable, robbing routers of the ability to distinguish flows at a finer granularity than *delegate*↔*destination* (Figure 5.3a). This may be too coarse-grained, especially since the flow ID is used for dropping packets from malicious flows. In effect, every flow that shares a delegate with a malicious flow will share its fate. (TE tends to work with coarser-grained flows, so destination addresses alone may be sufficiently granular.)



(a) Fingerprint Collection (b) Recursive Verification

Figure 5.4: **Briefing Techniques**

The simplest way to support finer-grain flow IDs is to include the delegate's SID in the calculation, similar to the way port numbers are used today. For example, delegates could assign a group of SIDs to each client source domain, which it can use to define network-level flows as it sees fit (see below). Accountability delegates with many clients would require a large pool of SIDs to achieve fine granularity (e.g., at the host or TCP flow level). If the SID is not sufficient, it is possible to add a separate flow ID field to the packet header to improve granularity. In our discussion, we will use the term flow ID to refer to both the SID only and SID plus dedicated field approaches.

Controlling Flow Granularity How the flow ID is assigned affects both privacy and the amount of collateral damage caused when an aggregate flow containing a malicious sender is blocked. At one extreme, a delegate could use a single flow ID for all its customers, which provides the biggest possible anonymity set, but may result in a lot of legitimate traffic being dropped if any client sends malicious traffic. At the other extreme, assigning senders unique flow IDs (Figure 5.3b), or a separate flow ID per TCP flow, allows fine grain filtering, but allows sender/TCP flow linkability. The solution we propose is for delegates to assign each client a pool of flow IDs which it can assign to packets based on internal policies. Delegates check that clients are using flow IDs assigned to them as part of the verification process (Section 5.5.3).

Source Domain Accountability Address Management An interesting alternative to senders picking a flow ID for each packet (within boundaries set by their delegate) is to have flow IDs assigned at the level of the source domain. For example, individual hosts could send packets with a traditional source address. If a packet leaves the source domain, the gateway routers replace it with an accountability address and hide the return address (like a NAT; see Section 5.6). This approach is especially attractive for source domains that act as the accountability delegate for their hosts (Section 5.7.4). Centralized management simplifies managing the pool of flow IDs, enforcing policies, and incremental deployment. The drawback is that individual users lose control over sender privacy.

5.5.2 Brief()

Accountability delegates need to know which packets their clients have sent if they are to vouch for them when challenged with a **verify()**. We consider two approaches in this section. Accountability delegates can choose any method, possibly on a per-client basis.

Fingerprint Collection. The simplest solution is for senders to proactively send their delegates fingerprints of the packets they send (Figure 5.4a). The delegate stores the fingerprints (e.g., for 30 seconds), and when it receives a **verify()**, it searches for the fingerprint and returns `VERIFIED` if it finds it. For reasons explained in Section 5.5.3, a packet's fingerprint is actually more than just a simple hash:

$$F(P) = H\left(K_{SD_S} \,||\, P_{header} \,||\, H(P_{body})\right)$$

Here H is a cryptographically secure hash function and K_{SD_S} is a symmetric key established when sender S signed up for service with delegate D_S. It is included in the fingerprint to prevent observers from linking P to $F(P)$. Each brief includes a client ID, a fingerprint, and a message authentication code (MAC):

Sender transmits packet and brief:

$$\begin{aligned} S \to R: &\quad P \\ S \to D_S: &\quad \textbf{brief}(P) = \text{clientID} \,||\, F(P) \\ &\qquad ||\, \text{MAC}_{K_{SD_S}}(\text{clientID} \,||\, F(P)) \end{aligned}$$

To reduce delegate storage requirements and network overhead, rather than sending full-sized fingerprints, hosts can instead periodically provide their delegate with a bloom filter of the fingerprints of all packets sent since the last brief. Accountability delegates keep the filters received in the last thirty seconds.

Note that in either case (fingerprints or bloom filters), the delegate can vouch for its clients' packets without knowing anything about their contents. Finally, if gateway routers assign accountability addresses, they can also take responsibility for briefing the delegate.

Bootstrapping Who vouches for briefs? That is, how do senders get briefs to their delegates if the packets carrying them cannot be verified? Clients include a special "token" in brief packet headers (e.g., as the SID in the destination address) proving to the delegate that the brief is from a valid client. Since verification requests include a copy of the unverified packet's header (see Section 5.5.3), the delegate can see that both the accountability *and destination* addresses point at the delegate, indicating the packet is a brief, cueing the delegate to check for the token. Delegates can use any scheme to select tokens. One possibility is using a hash chain based on a shared secret. Each brief uses the next hash in the chain, preventing replays. (This ensures the brief is from a valid client—we discuss "brief-flood" DoS attacks from malicious clients in Section 5.7.2.)

Recursive Verification. The alternative to fingerprint collection is to have hosts store the fingerprints of recently sent packets. When a delegate receives a **verify()**, the delegate forwards the verification packet to the host that sent it. The host responds "yes" or "no" to the delegate, which passes the response on to the original challenger (Figure 5.4b). In this case, **brief()** is a NOP. Recursive verification reduces network and storage overhead, but the catch is that in order

to work, each verification request must carry enough information for the delegate to map the packet to a customer. This impacts the flow ID granularity (Section 5.5.1): when using recursive verification, delegate must ensure that no two clients share a flow ID (or it must be willing to forward a verification to multiple clients).

5.5.3 Verify()

Verify() is nearly identical to AIP's anti-spoofing challenge, the difference being that an AIP challenge asks, "Is this packet's source address spoofed?" whereas **verify()** asks, "Do you vouch for this packet?" In AIP, first-hop routers periodically verify that packets purporting to be from a particular host are not spoofed. Likewise, in APIP routers periodically verify that flows are using valid accountability delegates and have not been reported for misbehavior. Verified flows are added to a whitelist whose entries expire at the end of each *verification interval* (e.g., 30 seconds); if a flow is still active, it is re-verified. Consider a sender S, a receiver R, and a router V ("verifier"). If V receives a packet P from S to R and the flow $S \rightarrow R$ is not in the whitelist, V sends a verification packet to S's accountability delegate, D_S (identified in the packet). To avoid buffering unverified packets, V can drop P and send an error message notifying S that P was dropped pending verification.

The verification packet includes P's fingerprint plus a MAC computed with a secret key known only to V. D_S now checks three things: (1) it has received a brief from S containing $F(P)$, (2) the accountability address in P is using an *SID* assigned to S, and (3) transmission from S to R has not been blocked via a shutoff (Section 5.5.4). If everything checks out, D_S returns a copy of the verification packet signed with its private key to V, which adds $S \rightarrow R$ to its whitelist. The protocol, below, shows both fingerprint collection ($\star$) and recursive verification ($\blacklozenge$), though only one or the other would be used in practice. (K_V is a secret known only to V; $K^+_{D_S}/K^-_{D_S}$ is the delegate's public/private keypair; K_{SD_S} is the symmetric key shared by S and D_S.)

Sender transmits packet and brief:

$S \rightarrow R$: P

$\star$ $S \rightarrow D_S$: **brief**(P)

Verifier sends error to sender and verification to delegate:

$V \rightarrow S$: `DROPPED (VERIFYING)` $\| F(P)$

$V \rightarrow D_S$: **verify**$(P) = P_{header} \| H(P_{body}) \| \text{MAC}_{K_V}(P_{header} \| H(P_{body}))$

Delegate verifies packet and responds:

$\blacklozenge$ $D_S \rightarrow S$: $\{$**verify**$(P)\}_{K_{SD_S}}$

$\blacklozenge$ $S \rightarrow D_S$: $\{$`VERIFIED` $\|$ **verify**$(P)\}_{K_{SD_S}}$

$D_S \rightarrow V$: $\{$`VERIFIED` $\|$ **verify**$(P) \| K^+_{D_S}\}_{K^-_{D_S}}$

V : add flow entry to whitelist

There are three points worth noting about this protocol. First, D returns the original verification packet so V does not have to keep state about pending verifications. V uses the MAC to ensure that it originated the verification request, preventing attackers from filling V's whitelist with bogus entries by sending it verifications it never asked for.

Second, the delegate needs to know the packet's destination address (R) so it can check if traffic $S \rightarrow R$ has been shut off. Since briefs only contain fingerprints, the delegate does not already have this information, so the verification request includes a copy of P's header. It also includes a hash of the body so the delegate can finish computing the fingerprint of packet being verified to check that it matches a brief in its cache.

Third, the last line in the protocol adds the flow to the white list, identified by its accountability address, destination address, and flow ID, as described in Section 5.5.1.

ISP Participation. Though *anyone* can verify a packet, APIP is most effective when routers closest to the source perform verification. An ISP that suspects a customer/peer might not be properly verifying its traffic can apply business pressure or possibly dissolve peering relationships if it finds an inordinate amount of unverified traffic. Another concern is that domains might verify traffic with a long verification interval (that is, after verifying a packet from a flow, the same flow is not verified again for an extended period of time). This allows malicious flows to do damage even if the flow's delegate receives a **shutoff()** since the flow will not be blocked until the next verification. The impact of long verification intervals could be mitigated if transit networks also verify traffic (see Section 5.8.1 for expected time-to-shutoff); after a **shutoff()**, the filter moves toward the sender as closer routers re-verify the flow. Also, even if routers are slow to react, APIP still facilitates a long-term fix eventually.

5.5.4 Shutoff()

Today, when hosts or routers identify a malicious flow, they can locally filter packets and work with neighboring ISPs to stop traffic. In APIP, they can also send a **shutoff()** request to the attacker's delegate. This is particularly useful for receivers, who should have the final say as to whether a flow is wanted or not. The protocol differs from AIP's shutoff protocol in two important ways. First, shutoffs are directed to accountability delegates, not to senders. Second a delegate can not only block the offending flow, but it can also pursue a long-term fix. The **shutoff()** protocol is shown below (between receiver R and S's delegate D_S regarding packet P from sender S):

Sender transmits packet and brief:

$S \rightarrow R$: P

$S \rightarrow D_S$: **brief**(P)

Receiver sends shutoff:

$R \rightarrow D_S$: **shutoff**$(P) = \{P_{header} \,||\, H(P_{body}) \,||\, duration \,||\, K_R^+\}_{K_R^-}$

Sender's delegate verifies shutoff and takes action:

D_S : check $H(K_R^+) == dest(P_{header})$

D_S : block offending flow for *duration* sec

Receivers can *always* shut off traffic directed at them. When the delegate receives the **shutoff()**, it checks that the **shutoff()** was signed by the private key corresponding to the recipient of the packet in question (so the **shutoff()** contains both the victim's public key and the original packet's header; the delegate compares the hash of the public key to the packet's destination address). If the verifier is a router and the **shutoff()** is signed by an ISP's key, it might also be honored, but perhaps

Adversary	End-to-end Encryption	Address Translation
Source Domain	Source domain always knows a packet's **sender**.	Source domain always knows a packet's **sender**.
Observers in Source Domain	Other **source domain customers**.	The **sender**. The sender's address is observable until the packet reaches the border router where NAT is performed.
Transit Networks	Starts as **source domain's customers** and **grows** the farther the packet travels. By the time it reaches the core, it could have come from anywhere.	**Source domain's customers**.
Receiver	The **sender**. The receiver decrypts the return address, which is the sender's address. If the sender is concerned with anonymity from the receiver, end-to-end encryption is not a viable option.	**Source domain's customers**.

Table 5.2: **APIP Privacy.** Comparison of sender anonymity set, as seen by different adversaries, for end-to-end encryption and NAT.

only with manual intervention—if a reputable ISP says one of your clients is attacking its network, chances are you should listen. After verifying a **shutoff()**, the attacker's delegate responds in two ways.

Short-term fix: To provide the victim immediate relief, the delegate blocks the offending flow by ceasing to verify packets from the attacker to the victim. Routers only save flow verifications in their whitelists temporarily; when a router on the path from S to R next tries to verify the attack flow, the delegate responds `DROP_FLOW`. This means the attack could last up to a router's verification interval—we discuss expected shutoff time in Section 5.8.1. If delegates work with ISPs, response time could be shortened by pushing verification revocations from delegates to routers. Alternatively, if we assume widespread shutoff support in NICs, delegates could send shutoffs to directly to attackers, as in AIP.

Long-term fix: Since clients sign contracts with their delegates, a delegate can contact the owner of misbehaving hosts out-of-band. Since most unwanted traffic comes from botnets of compromised hosts with well-intentioned owners [143], the owner will generally fix the problem, at which point the delegate can unblock flows from the host. If a client refuses to comply, the delegate can terminate service or report him to the authorities.

5.6 Masking Return Addresses

APIP separates accountability from other source address roles, allowing senders to hide the return address from observers in the network. APIP does not define any particular privacy mechanism, but rather enables various lightweight, "always-on" strategies for increasing the default level of privacy for all traffic without weakening accountability. We offer two examples: end-to-end return address encryption and network address translation. Since our focus is sender-flow unlinkability,

our primary concern is the size of the sender anonymity set from the perspective of four possible adversaries: the source domain, observers in the source domain, transit networks, and the receiver (Table 5.2).

End-to-end Encryption. Since the return address is used only by the receiver and not by routers, a simple idea is to encrypt it end-to-end (e.g., using IKE [127], à la IPsec); now only the destination and accountability addresses are visible in the network. We imagine two variants: one in which return address encryption is a network layer standard and can be performed end-to-end or gateway-to-gateway and one in which the return address is moved to a higher layer (e.g., transport or session layer).

Of course, though the return address is encrypted in the forward direction, it will be plainly visible as the destination address in responses; determined attackers may be able to link the outbound and inbound traffic (e.g., with timing analysis). Still, even this simple strategy offers increased privacy against passive observers, e.g., reviewing logs from a core ISP.

Network Address Translation. Encrypting the return address end-to-end hides it from the network, but not from the destination. For privacy from the network *and* the recipient, edge ISPs' border routers could perform address translation on outbound packets' return addresses by changing *NID:HID:SID* to *NID:HID':SID'*. (This can be done deterministically to avoid keeping large translation tables [191].) Note that in contrast to the encryption option, response packets sent by the destination will not reveal the identity of the original sender (in the destination address). The downside is that, in contrast to encrypted return addresses, the anonymity set shrinks closer to the source.

Today, increased use of NAT might be a controversial proposition, but cleaner thinking about source addresses mitigates some of the chief arguments against it. For example, in 2006 the entire nation of Qatar was banned from Wikipedia when one user vandalized an article because the country's sole ISP uses a NAT with one external IP address [32]; in APIP, all hosts could share one external return address while still being held individually accountable via the accountability address.

Second, NATs are traditionally deployed for address space separation—the privacy they provide is a side effect [216]. This is known to cause problems for servers or P2P applications. In contrast, we suggest NATing for privacy, which can be done selectively for outgoing connections. Incoming connections are not affected, so servers, for example, can publish their internal address to DNS and receive incoming connections without any kind of hole punching. Of course, NATing for incoming connections also has security benefits, but this is an orthogonal issue.

Reducing Overhead Not all packets need a return address. For connection oriented traffic, the return address needs to be sent to the destination only once during the establishment of the connection (and also when a mobile device switches networks). The destination can store it and reuse it for later packets. Doing so (1) ameliorates the header overhead introduced by splitting accountability and return addresses in the first place and (2) allows NATs to modify many fewer packets.

Beyond the First Hop No matter how far a packet travels, the sender anonymity set is still just

the sender's source domain. Though this may be sufficient for most senders, the NAT approach can be extended by performing address translation at more border routers. Though core ISPs are unlikely to do this, if the first 2–3 domains in the path do, a packet's sender anonymity set grows significantly before reaching the core (and ultimately its destination).

5.7 In the Real World

5.7.1 Holding Delegates Accountable

Delegates have three responsibilities: protecting the privacy of their clients, verifying packets with fingerprints that match those sent by valid clients, and dropping invalid packets. We briefly discuss how malicious or compromised delegates can either harm their clients or allow their clients to harm others.

(1) Releasing private information about clients. Delegates can learn a lot about who their clients communicate with, information they could use for their own benefit or reveal to a third party. Upon discovering a leak, the client can terminate service, find a new delegate, and potentially pursue legal action for breach of contract. Note that delegates only see packet headers, *not* packet contents. (An interesting direction for future work is exploring anonymous briefing schemes, e.g., based on cryptography or the use of an intermediary.)

(2) Failing to verify clients' packets. Delegates can effectively perform a DoS attack on clients by failing to verify their packets. Senders can detect this due to the excessive number of `DROPPED (VERIFYING)` or `DROPPED (VERIFICATION FAILED)` error messages they will receive from verifiers. Again, the client can then terminate service.

(3) Verifying invalid packets. Delegates can support attacks by verifying packets for which they did not receive a brief from a client or which belong to flows that have been shut off. (Such a delegate may be compromised or misconfigured or may even be colluding with an attacker.) Victims can detect such malicious behavior (e.g., by observing that their shutoff requests have been ignored).

Who Can be a Delegate? The likelihood of any of the above problems occurring depends on *who can be a delegate*. The issue of delegate oversight is complex; given space constraints, we can only hope to lay the groundwork for discussion and future work.

At one extreme, a single central authority could provide some form of oversight over delegates, similar to how ICANN accredits TLDs; verifiers would then only accept delegates on a whitelist published by this authority. This has the advantage that delegates can be monitored and misbehaving delegates can be immediately removed from the whitelist, creating an incentive for responsible delegate management. On the other hand, vetting all delegates is a huge burden for a single authority and the role (and power) of a single organization in charge of such a critical function is likely to raise political concerns.

The other extreme is a free-for-all: a host can pick any host to be its delegate. This flexibility opens the door for many deployment models. Besides commercial (third party) delegates, hosts can be their own delegates (similar to AIP), use their source domain as their delegate (Section 5.7.4), or form a peer-to-peer delegate network, in which hosts (or domains) vouch for one other. The critical

drawback of this flexibility is weaker protection against attacks—bots in a botnet, for example, can vouch for one another's packets regardless how many **shutoff()**s they receive. It will clearly be harder to defend against such attacks compared to the case where there are only a limited number of vetted delegates.

Naturally, a pragmatic solution likely falls somewhere in between these extremes. For example, a set of well-known commercial "delegate authorities" could emerge, similar to today's certificate authority infrastructure, each publishing a delegate whitelist. Alternatively, various groups could maintain delegate blacklists based on historical incidents, similar to today's security companies' publishing malware signatures. Individual verifiers can then decide which delegates to accept, a decision that could depend on many factors, including their position in the network (tier 1 versus edge), local regulation, historical information, or the "trust domain" they belong to [240]. Many other forms of semi-structured self regulation are possible.

5.7.2 Attacking APIP

We need to ensure that hosts cannot use APIP mechanisms to undermine APIP itself; two potential such attacks are "verification-flooding" and "brief-flooding."

Verification-Flooding. Attackers could attempt to overwhelm an accountability delegate with bogus verification requests—rendering it incapable of verifying honest hosts' packets—by sending a large number of dummy packets with accountability addresses pointing at the victim delegate. To these bogus verifications, the delegate could respond `DROP_HOST` (as opposed to `DROP_FLOW`). Source domains should track the number of `DROP_HOST`s their customers generate, taking action if it is too high.

Brief-Flooding. Similar to verification flooding, malicious *clients* could target their own delegate by sending a flood of briefs. This attack is tricky, since it is hard to distinguish from an honest host that happens to send lots of packets. Delegates can enact their own policies (e.g.: will accept 1 brief per second; must use bloom filters), which should be agreed upon when the client initially signs up for service.

5.7.3 Bootstrapping Trust

We now relax our initial assumption that HIDs are self-certifying; doing so does not break APIP, but requires us to do a small amount of extra work in **brief()**, **verify()**, and **shutoff()**.

brief() Clients already encrypt **brief()**s using a symmetric key established when the client registered for service, so no change is required.

verify() Delegates use their private keys to sign verification responses. If keys are not bound to HIDs, a PKI can be used instead; verifiers now need to check a delegate's certificate before trusting a response.

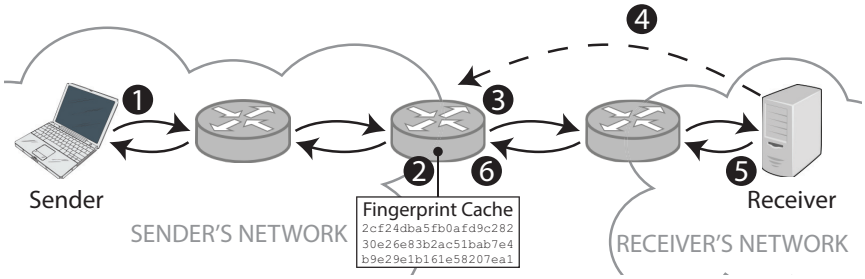


Figure 5.5: **Design 1:** ❶ Host sends packet using its own address as the source address. ❷ The ISP's border router saves a hash of the packet and ❸ performs address translation on the source address. ❹ If the packet is malicious, the receiver sends a **shutoff()** to the border router, otherwise ❺ it responds. ❻ The border router translates the response's destination address back to the original sender's address.

	Design 1	**Design 2**
Delegate	Source domain	Third party
Briefing	Fingerprint collect.	Recursive ver.
Source Addr.	Single field	Separate fields
Return Addr.	NAT	NAT or encrypt

Table 5.3: **Comparison of APIP Deployments.** Two possible instantiations of APIP.

shutoff() Victims sign **shutoff()** messages to convince the attacker's delegate that the shutoff truly came from the recipient of the offending packet. While we think it is reasonable to assume delegates register keys with a PKI for signing **verify()**s, there are many more hosts than delegates, so here we instead rely on verification. Upon receiving a **shutoff()**, the attacker's delegate sends a verification packet to the victim's delegate to check that the shutoff really came from the original packet's recipient:

$$\begin{aligned} R \to D_R: &\quad \textbf{brief}(\textbf{shutoff}(P)) \\ D_S \to D_R: &\quad X = \textbf{verify}^{\star}(\textit{shutoff}) \\ D_R \to D_S: &\quad \{\texttt{VERIFIED} \,||\, X\}_{K^{-}_{D_S}} \end{aligned}$$

When verifying a **shutoff()**, the delegate needs to look inside the shutoff packet, at the header of the original packet that prompted the shutoff, and check that its destination (the victim) sent the shutoff. (We denote **verify()** with this additional check **verify***().)

5.7.4 Concrete Designs

APIP is an architecture that allows routers and destinations to identify an entity that is willing to take responsibility for the packet, but properties of APIP depend on how it is deployed. For the sake of concreteness, we now sketch two end-to-end instantiations of APIP with very different properties. Table 5.3 summarizes the two designs.

Design 1. In the first design (Figure 5.5), the source domain acts as the accountability delegate for its hosts. Hosts are not aware of APIP and send packets with traditional source addresses. The

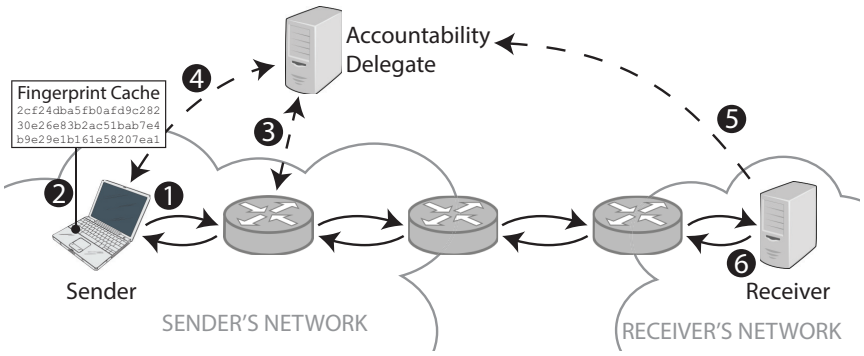


Figure 5.6: **Design 2:** ❶ Host sends packet and ❷ saves hash. ❸ First-hop router sends **verify()** to the packet's accountability delegate, which ❹ forwards the verification to the host. ❺ Using the accountability address, the receiver can send a **shutoff()**; otherwise ❻ it responds using the return address.

gateway routers for the domain use address translation to mask the return address, turning the source address into a combined accountability address and masked return address. They also collect packet fingerprints and respond to **verify()** and **shutoff()** requests. Gateway routers could either collectively act as a distributed accountability delegate and keep briefs locally, or they could periodically send briefs to a shared accountability server. This first design could be viewed as a variant of AIP, but implemented at the source domain level instead of individual senders.

This design offers a number of advantages. First, it is very efficient: gateway routers already naturally see all packets, eliminating the overhead of briefing a third party. Second, the source domain can immediately block malicious flows at the source in response to a shutoff, whereas external delegates can typically only stop flows indirectly. Third, hosts do not need to be modified.

Finally, this first design allows for incremental deployment of APIP over IP. Domains could implement accountability and address translation, as described. Packets would need to be marked to indicate whether the source domain supports APIP (e.g., by repurposing an ECN bit). Since both return traffic and **verify()**s/**shutoff()**s would arrive at the domain's border routers, **verify()** and **shutoff()** would each be assigned a new IP protocol number so the border router knows what to do with the packet. Since IP addresses are not cryptographic, external keys would have to be used to ensure the integrity of the **verify()** and **shutoff()** operations, as described in Section 5.7.3.

Design 2. The second design (Figure 5.6) uses a commercial third party that offers accountability delegation as a service (perhaps as part of a bundle with antivirus or firewall software). In this design, senders insert both an accountability address and a return address in the packet; the return address can be masked either with encryption or a NAT. Since the delegate is off-site, recursive verification is attractive: rather than regularly sending briefs, hosts save packet hashes and the delegate challenges clients when it itself is challenged.

One advantage of this solution is that it allows companies, universities, or small domains to avoid the hassle of managing delegate servers themselves by outsourcing delegation. Another advantage is that it is harder for observers in the network to determine what source domain the sender belongs to. The drawback is that there is more overhead than in the first design.

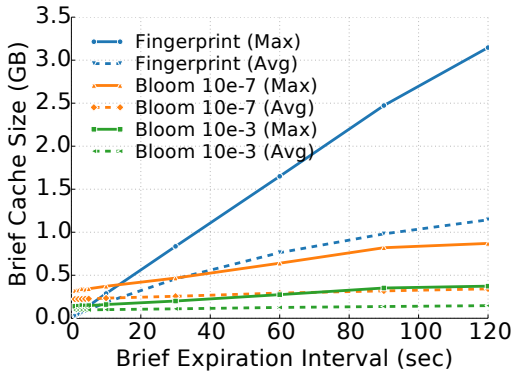


Figure 5.7: **Storage Overhead [brief()].** Brief cache size at delegate vs. fingerprint expiry interval.

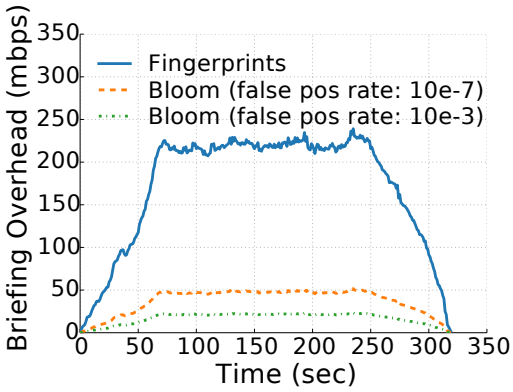


Figure 5.8: **Bandwidth Overhead [brief()].** Bandwidth required for briefs in our trace.

5.8 Evaluation

The primary question we consider in the section is: "is delegated accountability technically feasible?" Using a trace of NetFlow data from the border routers of a mid-sized university, we explore the costs of **brief()** and **verify()** and the efficacy of **shutoff()**. The five minute trace was taken on June 18, 2013 at noon and contains ten million flows. We then present a short privacy analysis.

5.8.1 Delegated Accountability

Brief() Briefing the delegate incurs computational overhead at the sender, storage overhead at the delegate, and bandwidth overhead in the network. In Section 5.5.2 we suggested that senders could report their traffic to their delegates by sending a list of packet fingerprints or by sending a bloom filter of recent fingerprints.

Computational Overhead Producing a packet fingerprint requires computing two hashes; we assume that computing the MAC of the fingerprint, in the worst case, costs one additional hash. Commodity CPUs can compute in the neighborhood of 5–20 MH/s [19], which translates to 0.9–3.4 Gbps (conservatively assuming 64B packets). This is more than reasonable for endhosts; for servers sending large volumes of traffic, we expect data centers could outsource briefing to an in-path appliance built with ASICs—current ASIC-based bitcoin miners perform 5–3,500 GH/s, translating to 0.9–600 Tbps.

Storage Overhead Next we consider the storage requirements at the delegate for saving briefs. Briefs are periodically purged from the cache; if a **verify()** arrives for a legitimate packet whose brief has been deleted, the sender must retransmit the packet. Assuming a single delegate serves all hosts in our trace, the first two series in Figure 5.7 show the size of the brief cache for different expiration intervals assuming the delegate stores individual fingerprints, each a 20 byte SHA-1 hash. The remaining series consider the space required if, instead of sending a fingerprint per

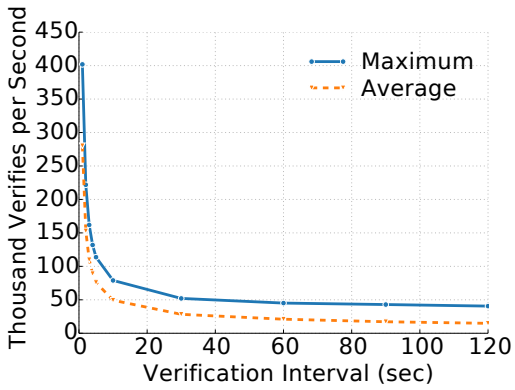


Figure 5.9: **CPU Overhead [verify()].** Verification rate at delegate vs. flow verification interval.

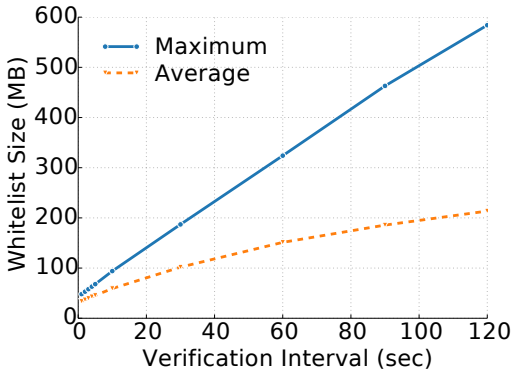


Figure 5.10: **Storage Overhead [verify()].** Size of verified flow whitelist vs. flow verification interval.

packet, hosts send a bloom filter every second for each active flow[1]. We assume that hosts size each filter appropriately based on how many packets from the flow were sent during the past second. If briefs expire every n seconds, we report brief cache sizes based on both the average and maximum number of packets seen across n-second bins in our trace.

Bandwidth Overhead Figure 5.8 shows the bandwidth required for the same briefing schemes discussed above. Sending fingerprints consumes an additional 2.5% of the original traffic volume, which was just under 10 Gbps; the bloom filter schemes consume 0.25%–0.5%. The bloom filter results assume a simple update scheme: every second, each host sends a bloom filter for packets sent in each flow[1] during the last 30 seconds; when the delegate gets a new filter for an existing flow, it replaces the old filter (using a bloom filter alternative that supports resizing is interesting future work). Note briefing overhead can be avoided entirely if (1) the ISP is the accountability delegate, so border routers save hashes directly, or (2) delegates use recursive verification (Section 5.5.2).

Verify() The magnitude of verification overhead is determined by the verification interval and flow granularity (fewer flows means fewer verifications; in our analysis, each "flow" is a TCP flow, so our numbers are an upper bound). There is a tradeoff between long and short verification intervals. The longer the interval, the more whitelist space is required at routers to remember which flows have been verified and the longer a malicious sender could transmit before being shut off. On the other hand, shorter intervals mean more work (more **verify()**s) for the delegate.

Computational Overhead Figure 5.9 shows how many **verify()**s per second an accountability delegate serving all the hosts in our trace would see at various verification intervals. A key observation here is that after 10 seconds, increasing the verification interval does not significantly decrease verification rate at the delegate since most flows are short. This knee suggests that 10 seconds might make a good interval for use in practice.

In our trace, a verification interval of 10 seconds generates a *maximum* of 78,000 **verify()**s per

[1]In practice, we think hosts should send one bloom filter for *all* traffic each second, not one per flow. Unfortunately, for privacy reasons, our trace did not include local addresses, so we could not merge flows originating from the same sender.

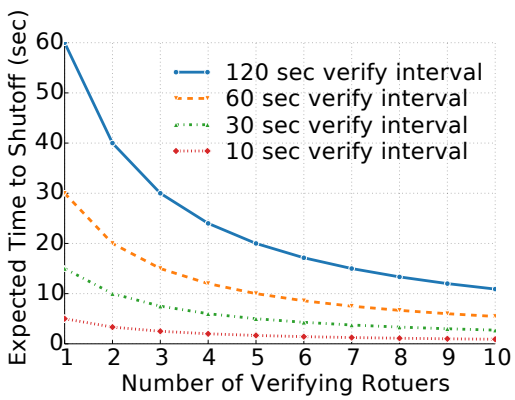


Figure 5.11: **Time to Shutoff.** Expected time to shutoff vs. number of on-path verifiers.

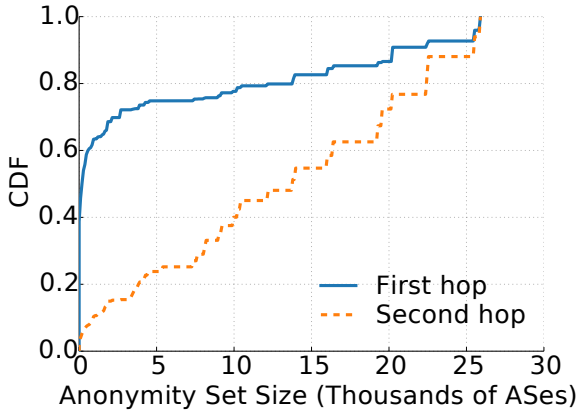


Figure 5.12: **Anonymity Set Size.** CDF of the number of customer ASes one and two hops down across all transit ASes.

second, each causing a lookup in a table with 1.5 million entries (assuming delegates save briefs for 30 seconds) and one signature generation at the delegate. We think these numbers are reasonable and leave enough headroom to comfortably handle larger networks—CuckooFilter [242] achieves more than 180,000 lookups per second in a table with *one billion* entries and the Ed25519 signature system [53] can perform 109,000 signatures per second on a 2010 quad-core 2.4GHz Westmere CPU.

Storage Overhead As routers verify flows, they keep a whitelist of verified flows so that every single packet need not be verified. Whitelist entries expire at the end of each verification interval, at which point the flow is re-verified. We use our trace to estimate the size of this whitelist.

To account for network architectures with addresses significantly larger than IP's, we assume each address is 60 bytes—20 bytes each for the *NID*, *HID*, and *SID*. (Many research efforts explore self-certifying IDs [40, 47, 125, 171] which are typically the hashes of a name or public key; we choose 20 bytes since SHA-1 produces 20 byte digests.) Entries identify flows at a host-host granularity, so each one is 120 bytes (two 60 byte addresses).

Figure 5.10 shows the size of the whitelist as we vary the verification interval. For a given verification interval, we group the flows in our trace into bins the size of that interval; flows belong to a bin if they were active during that time period (so a flow could belong to multiple bins). The figure reports the whitelist size based on both the average number of flows across bins as well as the maximum seen in any bin. A 10 second interval requires a maximum of 94 MB of whitelist space.

Shutoff() After receiving a **shutoff()**, a delegate blocks malicious flows by ceasing to verify them. The next time a router on the path from the attacker to the victim verifies the flow, the delegate returns `DROP_FLOW` and the router blocks the flow. How quickly this happens after a **shutoff()** depends on how many on-path routers perform verification and how often they verify each flow. Figure 5.11 shows the expected delay before a shutoff takes effect for different verification intervals as a function of the number of participating routers.

5.8.2 Privacy

How much privacy does APIP buy? If a sender uses its source domain as a delegate, this depends on the size of that domain. Raghavan et. al. find that, if ISPs were to aggregate prefixes geographically, over half of the prefixes advertised by many popular last-mile ISPs would include more than 10 million IPs [191].

If a sender uses a third party delegate, the anonymity set grows the farther a packet travels from the source domain. To see how much, we use Route Views [27] data from January 1, 2014 to roughly estimate AS "fanout." For each AS, we track how many customer networks sit beneath it in the AS hierarchy. (To be conservative, we only count an AS as a customer if it originates a prefix. Transit networks with no hosts do not contribute to an anonymity set.) For each BGP announcement, we add the origin AS to its first- and second-hop providers' customer sets. Figure 5.12 shows a CDF of first- and second-hop anonymity set sizes. Notably, 50% of ASes originating prefixes have at least 180 first-hop "siblings" and 90% have over 900 second-hop siblings. Though drawing conclusions about AS topology based on BGP announcements is imprecise, these ballpark figures give an idea of the anonymity benefits of delegated accountability.

5.9 Related Work

Privacy. Various techniques exist for hiding network source addresses, including crowds [193], mixes [75], and onion routing [192]. Real-world implementations based on these ideas include Anonymizer [4] and Tor [26, 96]. Liu et al. consider building onion routing into the network architecture itself [157]. NDN [139] takes a more radical approach by eliminating source addresses altogether; data finds the sender by following "bread crumbs" left by the request. The drawback to all of these approaches is a complete lack of accountability; there is no easy way to link malicious traffic with senders.

Raghavan et. al. [191] describe ISPs offering NAT for privacy as a service but uses a single source address. LAP [131] is similar to (but more secure than) our "NAT-at-every-hop" approach but does not consider accountability.

Accountability. Techniques like ingress/egress filtering [107, 148] aim to provide some degree of accountability by reducing the prevalence of source address spoofing; more sophisticated variants exist [98, 141, 185]. This class of approaches has limitations we address: (1) source addresses are only protected on a domain granularity, (2) filtering by itself provides no "shutoff" mechanism for misbehaving hosts who do not send spoofed packets, and (3) it is not compatible with schemes for hiding return addresses for the sake of anonymity.

As described in Section 5.3.1, our **verify()** and **shutoff()** mechanisms borrow heavily from AIP [40], which in turn based its mechanisms on ideas presented by Shaw [207] and in AITF [43]. By modifying these mechanisms to work with delegates, we make them privacy-preserving, enable long-term resolution, and avoid relying on self-certifying IDs.

Accountability delegates are described in [52], but the protocol is costly and is not evaluated; privacy receives only passing mention.

Balancing Accountability and Privacy. The idea of identify escrow is not new (e.g., [69]). In particular, our notion of delegated accountability is similar in flavor to the *contractual anonymity* described in RECAP [204], in which a service provider (e.g., an online forum) offers its users anonymity which can be broken only if they violate a pre-arranged contract. The key difference is that RECAP provides contractual anonymity at the application layer while we balance anonymity and accountability at the network layer, which poses unique constraints (like requiring source addresses to be both routable and anonymizable).

The closest work to ours is Persona [160], a network architecture designed to provide both anonymity and accountability. In Persona, routers re-encrypt source addresses at each hop to provide privacy; to provide accountability, the authors argue that by simply requiring each router to store all past encryption keys, a host under attack can ask the last-hop router to decrypt the packet's source address, revealing the second-to-last-hop router. This victim then contacts this router, repeating the process until the original sender is revealed. However, we argue that Persona does not really provide more accountability than the current Internet, but rather maintains the same level of accountability despite the improved anonymity. First, Persona does not enforce any anti-spoofing mechanism, so even if an attack victim asks the network to decrypt a source address, that address may be useless. Second, even if the initial source address is not spoofed, the recursive decryption procedure is less efficient than our approach, which just requires a communication with a single delegate.

Addressing. The use of addresses that consist of separate network, host and socket IDs, creating separate identifiers and locators, has been widely proposed [105, 142, 168]. [65, 71] discuss the meaning of source addresses, though without our focus on privacy and accountability.

5.10 Conclusion

This chapter attempts to show that a balance between accountability and privacy in the network *is* possible. By decoupling source addresses' roles as accountability addresses and return addresses, APIP strikes a balance between the two seemingly incompatible goals. Delegated accountability allows routers to verify that each packet they forward is vouched for and allows attack victims to report the abuse while at the same time permitting senders to hide their return addresses. Furthermore, the changes to traditional thinking about source addresses required to implement APIP are not radical; though more exploration is clearly required, we think the ideas presented here could be applied to the current Internet.

Chapter 6 Evaluating Network Layer Privacy Techniques

In Chapter 5 we introduced APIP, a network architecture that offers both better accountability and better privacy than IP does. And APIP is not the only proposal to make the network layer more private: a number of other alternative architectures promise improved privacy [77, 131, 139, 157, 158, 201, 217]. Naturally, we would like to be able to evaluate and compare these techniques, and so there are also many metrics for measuring "how private" a network architecture or protocol is. Unfortunately, in an effort to yield precise results, many of these analyses require specific information about the exact topology and state of the network; this makes these metrics hard to use and means that their results do not generalize well. Instead, in this chapter we consider what we call the ***architectural privacy evaluation problem***, which specifically focuses on measuring how private an architecture itself is, not a specific deployment of that architecture. We define a new privacy metric that does not depend on topology or traffic patterns and use it to evaluate architectures from practice and from research. We then explore a hypothetical "build-your-own" privacy service ISPs could offer to plug missing holes uncovered in our analysis.

6.1 Introduction

Online privacy is a critical issue today [23, 66, 68, 128, 182] and an important part of privacy is the ability to exchange messages across the network with some degree of anonymity—this is the third user goal (Keep Activity Private). While systems like Tor offer some degree of anonymity today at the expense of performance, researchers continue to explore how to update the network itself to protect users in the default case. In Chapter 5 we introduced APIP, a network architecture that offers both better accountability and better privacy than IP does. And APIP is not the only proposal to make the network layer more private: a number of other alternative architectures promise improved privacy [77, 131, 139, 157, 158, 201, 217].

Naturally, we would like to be able to evaluate and compare these techniques, and so there are also many metrics for measuring "how private" a network architecture or protocol is. Unfortunately, (1) these privacy metrics tend to depend on specific topology and traffic information that is hard to collect (like complete network topologies and instantaneous traffic patterns), and (2) even if you did collect it, the result would be very specific to that particular setting and would not generalize well. This means that, while these metrics do a good job characterizing specific scenarios, they are not well suited to evaluating architectures at a high level.

We focus on this latter problem, which we call the ***architectural privacy evaluation problem***: how do you make statements like "architecture 1 is more private than architecture 2" without the context of a particular topology or traffic pattern? Feedback at this level is important for network architects designing a new architecture or anonymity system—fundamental privacy problems in the architecture itself, like requiring a cryptographically verifiable source address in every packet [40], cannot easily be undone in the deployment or use stages. It is also important to users in actual operational networks, because, although there *is* a topology and a traffic matrix, users typically do not know it.

We propose a new measure of privacy, the *anonymity set radius*, based on how many hops between the sender (or receiver) and the adversary's vantage points are unknown to the adversary. This marks off a portion of the (unspecified) topology as the anonymity set, and the "radius" of this area serves as a proxy for its size (which we cannot report without information about a particular deployment). We use this idea to evaluate a handful of architectures from practice and from research. Finally, we imagine a new highly customizable, paid anonymity service ISPs could offer to help meet user needs not covered by today's anonymity systems. Our primary contributions are:

(1) A methodology for evaluating the privacy of an architecture outside the context of a particular deployment (Section 6.4). We define a new metric, the *anonymity set radius*, for characterizing an anonymity set without knowing details like topology or how many senders are in the network.

(2) An analysis of existing architectures in terms of the privacy they provide and at what cost (Section 6.4.2). In particular, we note conspicuous holes in the privacy vs. cost design space.

(3) The high-level design of a "build-your-own" privacy service ISPs could offer (Section 6.5). We describe how ISPs could offer anonymity as a paid service and let users customize exactly the privacy guarantees they need on-demand. We show how this service could fill the gaps we find in the design space in Section 6.4.2.

6.2 Network Layer Privacy

We are interested in the following ***privacy evaluation problem***: given a network protocol, architecture, or system (e.g., "IP with NATs" or "Tor"), evaluate it in terms of (1) *privacy* (how well does it work?) and (2) *cost* (what overheads are incurred to achieve that privacy?). To help us be more precise about the meanings of "evaluate," "privacy," and "cost," we start by discussing network privacy broadly.

A useful way to structure this discussion is walking through the workflow a typical network adversary would follow to break a user's privacy. In this chapter, we focus on property 4 from Figure III.4, Sender/Receiver Anonymity, which has three sub-properties: Sighting-Sender Unlinkability, Sighting-Receiver Unlinkability, and Sighting-Flow Unlinkability. This suggests that, at a high level, an attacker must (somehow) do the following:

1. **Construct per-sighting sender and receiver anonymity sets.** The adversary uses information from each sighting—like a source or destination address, or the topological location of the sighting—to identify *anonymity sets* for the sender and receiver *for that sighting* (i.e., sets of hosts who could have sent or received the packet). The larger the anonymity set the better.

2. **Link multiple sightings to a single flow.** The adversary tries to group all sightings from the same flow (either different sightings of the same packet—e.g., by matching identical payloads even if the headers have changed—or different packets in the same flow—e.g., packets with the same 5-tuple). Linking multiple sightings of one packet is important because different identity information might leak at different vantage points; if the adversary can link the sightings, it can combine these pieces of information. Likewise, different packets from the same flow may carry different identity information that could be combined.
3. **Combine information from the individual sightings.** Taking the intersection of all linkable per-sighting sender and receiver anonymity sets helps the adversary narrow down who the sender and receiver could be.

Thus, the evaluation techniques we propose focus on *quantifying what the adversary learns about the sender and receiver anonymity sets in some way*. This means we need a way to evaluate both (1) how much the adversary learns about the anonymity sets from each sighting and (2) the adversary's ability to link sightings.

There are already a number of proposed metrics for doing this. As we saw in Section III.3.2, an obvious example is simply reporting the size of the anonymity set. More sophisticated versions of this use probability or information theory to offer more meaningful scores (for example, assign each member of the anonymity set a "likelihood of being the sender" and take the entropy of this distribution). Unfortunately, these approaches suffer from two problems: (1) the information needed to compute a score is difficult to collect in practice—set size requires knowing how many hosts are in your network and assigning sender likelihoods requires very precise information about the topology and current traffic—and (2) even with the necessary information, each score is specific to a particular topology and traffic matrix; it is difficult to use these metrics to evaluate a architecture in general.

Let's make this idea more concrete. Networks can be described at different levels of detail, and the level you pick has a big impact on how you measure its privacy properties. The levels are, from high-level to low-level:

1. **Architecture:** The design of an architecture impacts its privacy in ways that generalize across multiple deployments. The primary signal to consider at this level is Packet Contents; header formats and forwarding semantics do not change with the topology. Topology comes into play only when an architecture sends a packet away from the shortest path to confuse the adversary. Likewise, Timing only enters this high-level picture if devices artificially delay packets (Section III.1.2).
2. **Deployed Network:** By "deployed network," we mean a concrete set of routers and hosts arranged in a particular topology. Topology now includes a notion of how many hosts are located in different parts of the network, so the location of a vantage point could reveal additional information not explicitly present in packet headers. Timing plays a slightly larger role at this level as well, since simple transmission and queueing delay calculations based on the topology can tell the adversary whether or not two sightings might be linked.
3. **In-Use Network:** When users are actually sending traffic across the network, we call it "in use." At this level, the Timing signal becomes more important, because the presence of cross-traffic has a big impact on the adversary's ability to link sightings to a packet or packets to a

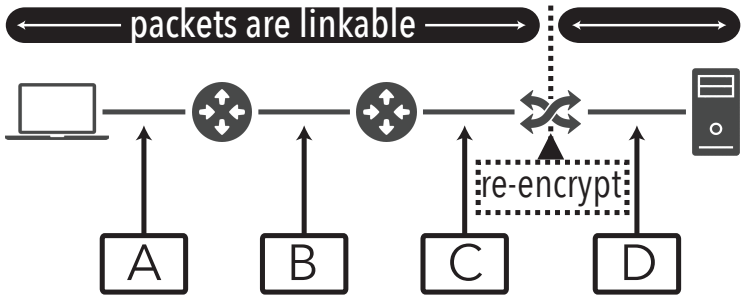


Figure 6.1: **Life of a Packet.** If packet headers are modified as a packet is forwarded, they might leak different information about the source and destination at each hop (represented here by *A*, *B*, *C*, and *D*). If they payload is also re-encrypted, then sightings before and after the re-encryption cannot be linked. Here, the adversary can combine information *A*, *B*, and *C*, but not *D*.

flow. If only one sender is active in the network, clearly nothing can provide them privacy.

The limitations we noted above about existing evaluation techniques boil down to this: those techniques are designed to assess deployed or in-use networks. Although measuring privacy at the deployed or in-use network level is more precise, the results are difficult to obtain in practice and do not generalize. Instead, we are interested in filling in what is, to our knowledge, missing: a privacy evaluation methodology at the architectural level. What are the properties of an architecture by itself, without the context of a particular deployment? This is useful for network architects in the design phase and network operators in the deployment phase who want to understand the impact of their design on privacy.

This means we are largely concerned with how the architecture operates on a packet rather than how one sender's traffic interacts with another's. Consider a packet moving through a network from the perspective of privacy: the packet traverses a series of devices, some of which modify its contents, timing, and path in an effort to hide the Packet Contents, Timing, and Topology signals (examples include re-encrypting the payload or exposing a previously encrypted next-hop header). In this chapter, we call these types of modifications **privacy primitives**; they were discussed in detail in Section III.1.2 (techniques 4–10) and are summarized here in Table 6.1. The primitives (1) cause packets to leak different information about the sender and receiver at different points along its path (represented abstractly as *A*, *B*, *C*, and *D* in Figure 6.1) and/or (2) prevent an adversary from linking sightings of the packet in one part of the network from another. These points form the boundaries of what we call *linkable segments* (there are two in the figure)—the adversary knows that all sightings within a linkable segment belong to the same packet, so the information leaked by these sightings can be pooled.

6.3 Method 1: Share Count Analysis

Our first technique, *share count analysis*, draws inspiration from header space analysis (HSA) [146], though the details are different since our goal is not to verify properties of *actual operational networks* but rather to *quantitatively compare network architectures*. Like HSA, we send test packets through models of network devices and track how the headers change. Unlike HSA, we are not

Primitive	Description	Costs	Examples
Indirection	Forward packet along a path other than the default path to the destination. *(E.g., send packet through a series of overlay nodes.)*	(1) Latency overhead due to path stretch. (2) Bandwidth overhead for sending packet on more links than necessary.	Tor [96]
Artificial Delay	Gather a batch of packets, wait, and release them in a different order.	(1) Latency overhead due to batching and delay.	Mixes [75]
Dummy Traffic	Inject artificial cross-traffic.	(1) Bandwidth overhead for sending extra packets.	CSL [239] HISP [199]
Mutate Payload	Change packet contents to prevent linking multiple sightings of the same packet. *(E.g., re-encrypt the packet at each hop.)*	(1) Encryption computation. (2) Key management/distribution.	Tor [96]
Encrypt State	Carry state in packet header individually encrypted for each device. *(E.g., source routing with each entry encrypted for the corresponding router, like how Tor builds circuits.)*	(1) Encryption computation. (2) Key management/distribution.	Tor [96] LAP [131]
Store State	Store state on the devices that need it, not in headers. *(E.g., a NAT stores a packet's source address and removes it from the packet itself.)*	(1) Per-flow storage. (2) Per-packet access latency. (3) Fate sharing: if state is lost, flow is lost.	NAT NDN [139]
Multicast	Omit some destination entirely. *(E.g., omit destination and broadcast packet to all hosts or include only destination domain and multicast to all hosts in that domain.)*	(1) Bandwidth overhead for sending multiple copies of packet. (2) Computation (each host needs to check if it is the true destination for each packet).	Anocast [36] WAR [64]

Table 6.1: **Privacy Primitives.** General techniques for hiding input signals. Primitives in the top section hide topology, those in the middle section hide timing, and the bottom group hide packet contents. Each one incurs some kind of overhead. These ideas are used as building blocks to construct privacy-preserving protocols. Note that scrambling packet contents does not directly hide communication state, but rather prevents the adversary from putting together communication state learned at different vantage points.

concerned with the actual bits in the header, but rather how much information they give away. We represent this leaked information using *share counts*, which we explain in Section 6.3.1. First we describe our models for the adversary, packets, and the network, and then we describe how we use share counts to analyze an architecture.

6.3.1 Methodology

Network Model

Packets. To address diversity in header format, we make no assumptions about the contents or formats of headers. Instead, we represent each packet as four *meta-fields*: all of the header bits that contribute to identifying (1) the sender, (2) the source network, (3) the destination, and (4) the flow. These meta-fields may overlap—for example, in IP, the source address both identifies the sender

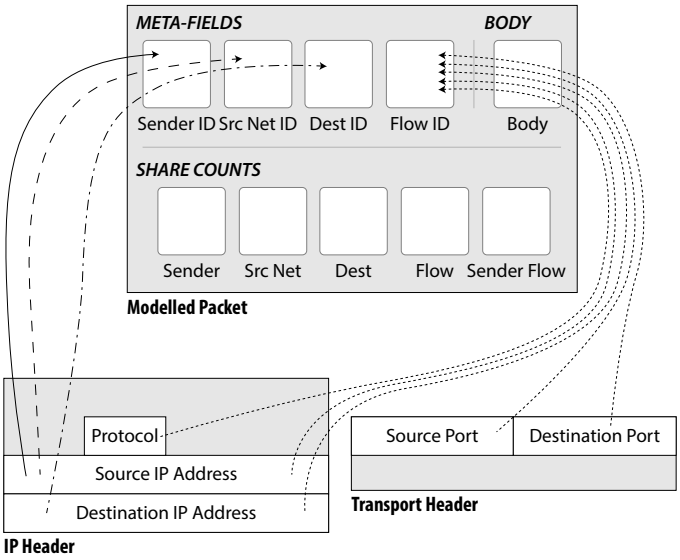


Figure 6.2: **Meta-fields.** Example showing how TCP/IP fields map to meta-fields.

and contributes to identifying the flow. Before using our tool, a human expert must decide how to map the bits in each architecture's header to these four meta-fields. (Figure 6.2 shows this mapping for TCP/IP.)

In addition to the meta-fields, each packet has a *body*, which, as noted above, we assume is encrypted and so we treat as an opaque value used only for equality testing to link multiple snapshots of the same packet together even if the headers have changed.

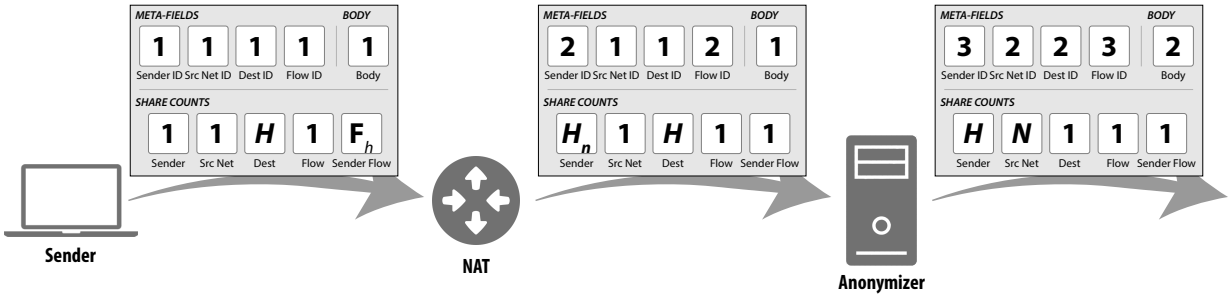
Finally, each packet carries a *share count* for each meta-field. A share count indicates how many entities in the network could share the current value for that meta-field. For example, how many senders share the same value for the Sender-ID? In IP, when a packet leaves the sender, its Sender-ID share count is 1; when it leaves a NAT, the share count is the number of hosts in the source network. We also include a fifth share count, which tracks how many of a sender's *flows* share the same Sender-ID (as opposed to how many *senders*). For instance, if a host is multihomed and sends half of its traffic on each link, the sender flow share count is $\frac{1}{2}$.

Network Boxes. We model a path through the network as a series of *network boxes* connected by links. Boxes may change the values of some header fields (and therefore change the values of some meta-fields and share counts) or of the body (e.g., a Tor relay re-encrypts it). Each box is defined by two properties (which must be manually specified for each architecture):

1. Does the box change any *meta-fields* and/or the *body*? We do not model actual values; each field is represented as an integer, which is incremented if a box changes it.
2. What are the *share counts* for each meta-field after a packet leaves the box?

Analysis Procedure

Share count analysis uses the model presented above by creating a path of network boxes, labelling some links and boxes as vantage points, sending a symbolic packet along the path, and recording



(N = num networks; H = num hosts; H_n = hosts per network; F_h = flows per host)

Figure 6.3: **Share Counts Example.** Test packet travelling through a NAT and an anonymizer.

the share counts seen by the adversary. The adversary learns something if a share count ever reaches one. (Being able to model an adversary simply by placing vantage points on any set of links and boxes gives us the flexibility to represent a variety of adversaries—we discuss this further below.) Here we describe the details, presented in five steps.

(1) Generate paths. Since we do not currently take the physical location of a vantage point into account, we test using a single path. The base path is:

```
[sender]-[router]-[destination]
```

When we test an architecture or tool that requires specialized boxes, we add those boxes as needed. For example, the test path for IP w/ NAT is:

```
[sender]-[nat]-[router]-[destination]
```

Depending on the adversary we want to model, we set certain links and boxes to be vantage points (see Section 6.3.1).

(2) Forward test packet. Next we forward a test packet through the path, updating the meta-fields, body, and share counts at each box (Figure 6.3). At each vantage point, we save a snapshot of the packet state.

(3) Group "linkable snapshots." The adversary uses the packet body to link packet snapshots from different vantage points. If a box changes the body, then, as far as the adversary knows, snapshots from vantage points on either side of that box were generated by different packets, meaning the adversary cannot combine the information it learns from each set of snapshots individually. Of course, if the adversary has a vantage point *on* a box that changes the body, then it *can* link the snapshots. The result of this step is one or more sets of *linkable snapshots*.

(4) Consolidate information from snapshots. For each set of linkable snapshots and for each meta-field, select the snapshot with the minimum share count. Record this share count and the corresponding meta-field value. For the Flow-ID share count and the sender flow share count, we instead record the maximum.

This network info...	...narrows the anonymity set to...	...if the adversary knows
Source domain	customers of source ISP	list of ISP's customers
Source domain	affiliates of business/university	employee/student roster
Source domain	residents of neighborhood	network address to location mapping
Destination domain	customers of company/service X	list of X's customers
Destination domain	people interested in topic Y	advertising profiles
Destination domain	particular user of public site Z	activity log (posts & timings) from Z

Table 6.2: **External Information.** Examples of external information an adversary could use to connect packets to human users.

(5) Test what adversary learned. We can use the minimum share counts to test whether the adversary achieved its goals. It was able to link the source network to the destination (G1) if:

(Source-Network-ID share count == 1) &&
(Destination-ID share count == 1)

Similarly, the adversary linked the sender to the destination (G2) if:

(Sender-ID share count == 1) &&
(Destination-ID share count == 1)

Metrics

Share count analysis allows us to evaluate architectures using multiple metrics, each involving a run of the procedure described in Section 6.3.1. In this evaluation, we define two specific goals for the adversary:

(G1) **Given a packet, link the human sender to the destination *("WHO").*** We assume that the adversary's best shot at doing this is to learn the packet's source network, which, combined with some amount of external information (see Table 6.2), could yield a small set of human individuals.

(G2) **Given a user, construct a history of their online activity *("WHAT").*** For this, the adversary must link flows to a common "sender ID" (e.g., an IP address). The sender ID may be opaque in the sense that it does not identify a person; here it is only needed to link flows from the same (potentially unknown) person.

We start by asking whether four real-world adversaries can achieve G1 and G2:

1. **Local Eavesdropper** *(e.g., someone sniffing open Wi-Fi)*: We place a vantage point on the link leaving the sender.
2. **Global Surveillance** *(e.g., monitoring by a nation-state)*: We place a vantage point on every router.
3. **Source ISP** *(e.g., employer or school)*: We place a vantage point on the link leaving the sender. We relax G1 and G2 to only check the Destination-ID share count, since the source ISP already knows the sender and source network.

	1-G1	1-G2	2-G1	2-G2	3	4-G1	4-G2	5-G1 (6-G1)	5-G2 (6-G2)	7	8
IP	•	•	•	•	•	•	•	1 (3)	1 (3)	1	F_h
IP-NAT	•	•	•	○	•	•	○	1 (5)	1 (2)	1	F_h
IP-Tor	○	○	○	○	○	○	○	3 (1)	3 (1)	1	F_h
APIP-ISP-NAT-Unique	•	•	•	•	•	•	•	1 (5)	1 (5)	1	$\frac{F_h \cdot H_d}{K}$
APIP-External-Encrypted-Shared	○	○	○	○	•	•	•	∞ (0)	∞ (0)	H_f	1
i3	○	○	○	○	○	○	○	1 (1)	1 (1)	1	F_h

(F_h = flows per host; H_d = hosts per delegate; K = num flow IDs; H_f = hosts per flow ID)

Table 6.3: **Comparing Architectures with Share Counts.** Overall comparison of representative combinations of architectures and tools.

4. **Destination** *(e.g., web server)*: We place a vantage point on the destination. We relax G1 and G2 to only check the Source-Network-ID and Sender-ID share counts, since the destination already knows its own identity.

Next, we consider arbitrary network adversaries by testing all combinations of vantage points and asking:

5. What is the minimum number of vantage points needed to achieve G1? And G2?
6. How many different combinations can achieve these minimums? (More possibilities means more opportunities for the adversary to succeed.)

Finally, we check two adversary-independent metrics:

7. What is the maximum Flow-ID share count at seen at any box?
8. What is the maximum sender flow share count seen at any box?

For privacy reasons, some architectures may give multiple TCP flows the same Flow-ID. However, this means that network boxes have coarser-grained handles for traffic engineering and, worse, that if an administrator wants to block a misbehaving flow, other benign flows will be blocked with it. Therefore, the maximum Flow-ID share count (7) is a measure of collateral damage. If a single host has multiple Flow-IDs, the maximum sender flow share count (8) indicates how successfully the adversary can reconstruct a user's activity history.

Finally, note that the eight questions we list here are merely examples of the kinds of metrics share count analysis supports; as we extend the model, we can ask more (and more sophisticated) questions. As it stands, though, this list demonstrates the variety of information we can learn from share counts.

6.3.2 Results

We implemented share count analysis and ran it on IP, APIP, and i3. We tested five variants of IP (IP, IP w/ NAT, IP w/ Anonymizer, IP w/ Tor, and IP w/ NAT and Tor). We assume i3 is deployed as an overlay where packets from endpoints to the i3 infrastructure expose the endpoint's IP address and that TLS is used between endpoints and the i3 rendezvous node, meaning the payload is re-encrypted at the rendezvous server.

APIP packets carry two addresses: a *return address* (for replying to the sender) and an *accountability address* (which points to a third party delegate who fields complaints about the packet). We tested all eight combinations of the following three options: First, senders can "hide" their return

address using (1) **NAT** or (2) **encryption** (the return address is encrypted with the payload, so it is only accessible to the final destination). Second, the delegate could be run by (1) the **source ISP**, meaning the accountability address gives away the sender's source network, or (2) an **external third party**. Third, each APIP packet contains a flow ID, which routers use to block malicious flows. Delegates assign each client either (1) a set of flow IDs that are **unique** to it or (2) a set of flow IDs that are **shared** among multiple of the delegate's clients.

In this section we summarize the results, which are labelled by question number (1–8) and goal (G1 or G2) (see Section 6.3.1). For yes/no questions, • = yes and ◦ = no.

Comparing Architectures. To start, Table 6.3 presents the complete results for IP, IP w/ NAT, IP w/ Tor, a weak and a strong variant of APIP, and i3.

First, we see that IP was not designed with privacy in mind; you need Tor to get any kind of privacy guarantees. Using a NAT helps a little by preventing adversaries without vantage points inside the NAT from linking two separate flows initiated by the same sender. Tor does quite well, although this analysis does not reflect its performance overhead.

Second, the weakest version of APIP (ISP delegate with NATed return addresses and unique flow IDs) appears to be no better than plain IP. There is one subtle difference in column 8, however: since each sender has a set of flow IDs to choose from for each flow, the adversary can only link a fraction of a sender's activity. (Furthermore, though not represented in the results, any version of APIP has stronger accountability than IP.)

Next, the strongest version of APIP we tested falls short of Tor in terms of privacy, but it comes with none of Tor's performance costs. Also, note that the infinities for 5-G1 and 5-G2 are misleading; currently our model only considers on-path boxes, so infinity is correct in the sense that there is no combination of on-path boxes that could be compromised to achieve G1 or G2. However, if the accountability delegate were compromised, these connections could be made.

Finally, i3's stats match Tor's, with the exception that fewer vantage points are needed to achieve G1 and G2 (the adversary only needs to compromise the i3 rendezvous node instead of three Tor relays).

When Can the Adversary Learn the Source Network? The columns in the table below indicate whether a local sniffer, global surveillance, and the destination can learn the source network, respectively.

	1-G1	*2-G1*	*4-G1*
IP	•	•	•
IP-NAT	•	•	•
IP-Tor	◦	◦	◦
APIP-ISP--**	•	•	•
APIP-External--**	◦	◦	•

Since IP was not designed to protect this information, only Tor is able to hide it; even a NAT does not help. For APIP, if source domains act as accountability delegates, then the accountability address gives away the source network. With an external delegate, the source network is hidden from local and global adversaries. However, the destination server still learns the source network. If the return address is included (encrypted) inside the packet body, then the destination learns

the sender's unmodified address and therefore the source network. If the return address is hidden using NAT, then the destination cannot connect the packet to a particular sender, but still learns the source domain. This could be mitigated if multiple ISPs were willing to perform address translation as packets leave their networks.

Finding Effective Vantage Points. Share count analysis can give us a sense of *how hard* it is for an adversary to achieve a particular goal by determining the minimum number of vantage points needed (fewer vantage points means easier attack) and how many ways those vantage points could be placed (more options means easier attack). For example, in IP a single vantage point is enough to link both the sender and the source network to the destination. With Tor, on the other hand, three vantage points are needed (the three Tor relays). And, with Tor and a NAT, linking the sender to the destination requires a fourth vantage point: one inside the NAT to link the sender's internal and external addresses.

	5-G1	*6-G1*	*5-G2*	*6-G2*
IP	1	3	1	3
IP-Tor	3	1	3	1
IP-NAT-Tor	3	1	4	2

Of course, share count analysis can also produce the successful paths themselves. As a sanity check, we see that for IP with NAT and Tor, the adversary must compromise the Tor relays and also either the link inside the NAT or the NAT itself (∗ indicates a vantage point):

```
[sender]*[nat]-[router]-[tor-entry*]-[router]-[tor-relay*]-[router]-[tor-exit*]-[router]-[destination]
[sender]-[nat*]-[router]-[tor-entry*]-[router]-[tor-relay*]-[router]-[tor-exit*]-[router]-[destination]
```

The Cost of Privacy. Next we consider the adversary's ability to link multiple flows to the same user for the different variants of APIP. The first column in the table below shows whether a global adversary can link the packet to both the destination and the sender (2-G2). The second column shows how many total TCP flows are shut off when a misbehaving flow is reported (in APIP, routers block bad flows that have been reported to accountability delegates). Finally, the third column shows how many of the sender's flows can be linked (in APIP, each sender is given a group of flow IDs to assign to its flows as it chooses).

	2-G2	*7*	*8*
APIP--*-Unique*	•	1	$\frac{F_h \cdot H_d}{K}$
APIP--*-Shared*	◦	H_f	1

(F_h = flows per host; H_d = hosts per delegate;
K = num flow IDs; H_f = hosts per flow ID)

First, notice the obvious tradeoff between sender-flow linkability and collateral damage. With shared flow IDs, the adversary cannot link flows to users, so it cannot build a history of any user's online behavior. On the other hand, since multiple hosts share the same set of flow IDs, when a router blocks a malicious flow, it could also block up to H_f benign hosts' flows as well.

Second, even when the adversary can link flows to senders (i.e., when each sender is assigned a set of unique flow IDs), since it has multiple flow IDs to choose among, traffic sent with different flow IDs appears to come from different senders. In the third column, we see that the maximum

fraction of the sender's flows that can be linked is $\frac{F_h \cdot H_d}{K}$. The denominator, K, is the number of possible flow IDs, which is determined by the number of bits in the header given to the flow ID. This is useful feedback to protocol designers, who can now see a direct numerical link between privacy and header format.

6.4 Method 2: Anonymity Set Radius

The advantage of the share counts approach is its ability to answer a variety of questions. For instance, as we just saw, share count analysis was able not only to tell us when an adversary was able to learn the identity of the sender and/or receiver, but also how many vantage points were required and which combinations of vantage points were successful. Unfortunately, the share count abstraction does not perform as well as we would like in the absence of a concrete topology or external information (Table 6.2). This is because, without concrete numbers, it lacks resolution, so to speak. The Sender-ID share count can be `ONE_HOST`, `ONE_NETWORK`, `MANY_NETWORKS`, or `WHOLE_INTERNET`; we would like more granularity. In this section, we propose an alternative metric for quantifying network architecture privacy called the **anonymity set radius**. While less expressive than share counts, it offers better discerning power with incomplete information.

6.4.1 Methodology

At a high level, our methodology can be summarized as follows:

INPUT: (1) An architecture, specified as a list of privacy primitives and where they are used. (2) An adversary, specified as a set of vantage points.

OUTPUT: (1) The most specific information the adversary can learn about the sender and receiver of a flow. (2) The total cost of the primitives used.

METHOD: (1) To evaluate privacy, we first identify the linkable segments between the sender and receiver and then characterize what the adversary learns about the sender and receiver anonymity sets in each segment. The result will be a list, with one entry per linkable segment, indicating what the adversary learned about each region. In Figure 6.1, this would be $[A + B + C, D]$. (2) To evaluate cost, we associate one or more costs with each primitive (e.g., storage overhead, header overhead, or processing overhead) and report the number of times each cost is incurred (e.g., "state stored on n routers").

The remainder of this section explains our approach to evaluating architectural privacy:

1. We propose definitions for "privacy" and "cost" at the architectural level.
2. We present a methodology architectural privacy evaluation.

Characterizing Privacy

In this section, we propose a way to quantify an anonymity set *without knowing a concrete topology or distribution of hosts* (recall that metrics like set size require knowing the topology and host count, and entropy additionally requires traffic information). As a proxy for anonymity set size, we

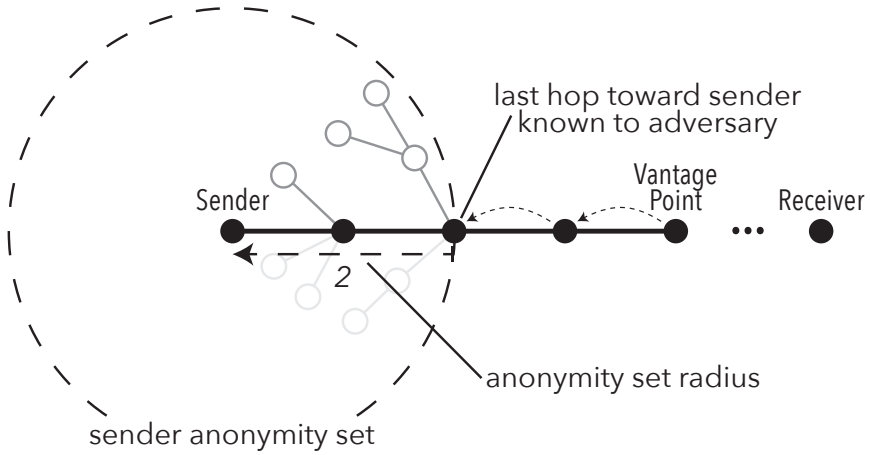


Figure 6.4: **Anonymity Set Radius.** Without a concrete topology, we measure the size of the anonymity set in hops, which give an intuitive, though not exact, sense of its size. Nodes could represent routers or ASes; solid black nodes are on the path from sender to receiver, while hollow gray nodes are other routers or ASes in the anonymity set.

introduce *anonymity set radius*. Though less precise than entropy, it does not require a topology or traffic matrix.

Anonymity Set Radius. To represent the size of an anonymity set when we do not know the topology or how many hosts or ASes are in the network, we count the number of "hops" (at the granularity of routers or ASes) from the sender to the closest point on the path that the adversary can identify as a proxy for the anonymity set size. We call this the *anonymity set radius* because you can think of the distance between a point on the path and the sender as defining the radius of a circular sender anonymity set (see Figure 6.4). Picture the path from a vantage point back to the sender and suppose the adversary were trying to physically move along this path to the source—how many hops could it take before it no longer knows where to go? This point marks the edge of the portion of the network that could contain the sender. The "radius" of this area is the number of remaining hops from the border to the source (in the figure, 2). The same applies to the receiver. The term "radius" implies that exposing a linear number of hops may result in a more-than-linear decrease in anonymity set size, because learning each successive hop to the sender eliminates subtrees of ASes that could have contained the sender.

For example, if the adversary sees a packet with a public IP source address, it knows the exact sender, so the anonymity set is 1 host and has a radius of 0. If the sender is behind a NAT, the adversary only learns the sender's network, so the anonymity set is the whole source network and the radius is 1. And, in the extreme case of Tor, the radius is infinity—that is, the anonymity set includes all hosts in the Internet—when a packet is between two Tor relays, because both the packet's headers and physical location give away nothing about the true sender or true receiver.

Radius Reduction. What is a "good" radius? When should a network architect conclude their design is "private enough"? It is tempting to say headers should leak *no identifying information*—the radius, from observing just Packet Contents, should be infinity! Though an appealing goal, this

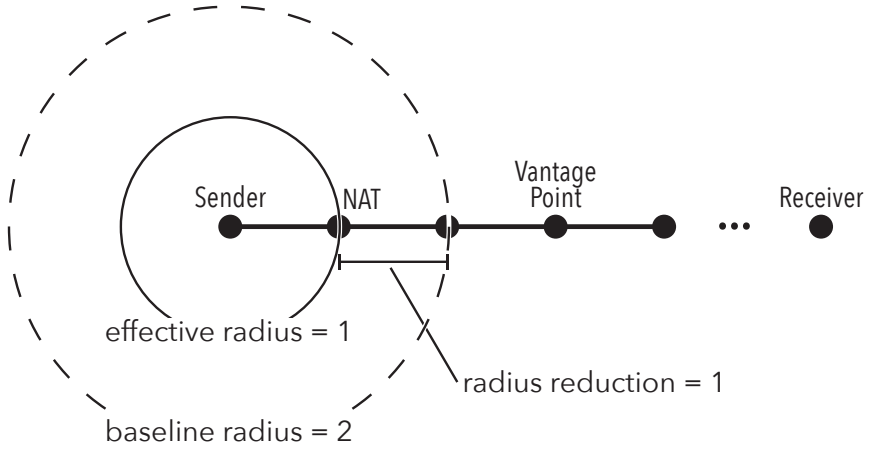


Figure 6.5: **Radius Reduction.** A packet's physical presence at a particular vantage point gives away a baseline amount of information about the anonymity set (dashed circle). Information in packet headers might allow the adversary to narrow this down to a smaller *effective radius* (solid circle). In this example, the NAT replaces the sender's address with the network's address, which yields an effective radius of 1. Since the baseline radius is 2, we say the packet headers caused a *radius reduction* of 1.

is not realistic because some amount of information about the sender and receiver is necessary for communication to happen (see the discussion of communication state in Section III.1.2). Fortunately, protocol designers can reasonably let Packet Contents leak more than zero because some information is already leaked by Topology and Timing. A packet's mere presence on a link, even if all the bits in the packet were set randomly, gives away some amount of information, which we call the *timing and topology (T&T) baseline:*

> **Timing & Topology Baseline:** *The information leaked by the packet's physical presence along the default path from sender to receiver even if all the bits in a packet were set randomly.*

This is the baseline amount of leakage to be expected in a typical network communication; the primitives for hiding the timing and topology signals (Section III.1.2) typically require taking a longer path or introducing artificial delay—overhead users only want to pay when privacy really matters. A reasonable target for protocol designers, then, is to ensure that, in the default case, headers never leak more than the T&T baseline already does.

To measure this, we introduce a second metric: *baseline radius reduction*. The radius by itself captures how much the adversary is able to narrow down the anonymity set; radius reduction captures how much additional information packet headers leak. Since each router knows which port a packet arrives on and which port it leaves from, Topology gives away the previous and next hops in the default path from source to destination; communication state should not give away more than this. Radius reduction captures how many extra hops are revealed in reality.

As shown in Figure 6.5, the physical location of a vantage point results in a *baseline radius* (the dashed circle). Information from packet headers could give the adversary more information about the source, leading to a smaller *effective radius* (the solid circle shows an example for vanilla IP with a NATed source address—the source address shrinks the anonymity set to the source network, with a radius of 1). Since the baseline radius was 2, the radius reduction due to headers is 1.

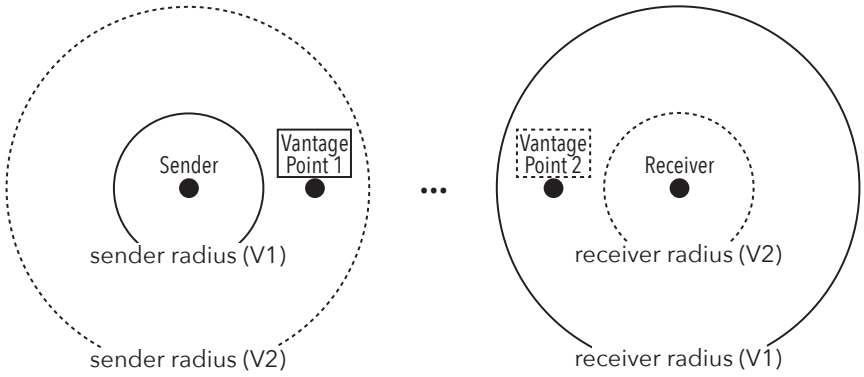


Figure 6.6: **Multiple Vantage Points.** The adversary might learn different information at different vantage points. For example, vantage points near the sender typically learn more about the sender than the receiver; vice-versa for vantage points near the receiver (we see this in the anonymity radii shown in the figure). If the adversary can link sightings from both vantage points, it can learn a lot about both the sender and the receiver.

Characterizing Cost

To characterize the cost of an architecture's privacy features, we again turn to the privacy primitives in Section III.1.2. Each primitive incurs one or more overheads, listed in the third column of Table 6.1. There are five general types of cost: (1) **communication latency**, which impacts the user; (2) **bandwidth overhead**, which impacts both the user (throughput and data plan usage) and the network (provisioning for congestion); (3) **storage**, which requires space on network devices, adds access latency during forwarding, and can break ongoing connections if state is lost; (4) **computation** for cryptographic operations, which could lower device throughput; and (5) **key distribution**, which introduces management burden.

To characterize an architecture, we count the number of devices (e.g., routers) employing each primitive, adding a tally to each corresponding overhead category; we then report this list of counts. For example, the path in Figure 6.5 has a NAT, which stores the source's address, so the cost is *[1x Storage]*. Though this is a loose, qualitative characterization, it gives a sense of privacy vs. cost tradeoff an architecture makes.

Evaluating an Architecture

At a high level, the idea is to first determine the sender and receiver radii for each vantage point (based on the vantage point's location, information in the headers, and whether or not the packet leaves its default path) and then to check whether or not sightings at those vantage points are linkable (based on information in the headers, whether the payload has been altered, and whether or not the packet is artificially delayed). Typically one vantage point learns a smaller sender radius, another learns a smaller receiver radius, and the adversary would like to put the two together (Figure 6.6).

Evaluating with a Path. Suppose we want to evaluate an architecture. To start, for simplicity, assume we are given the complete path of devices from sender to receiver—not necessarily a

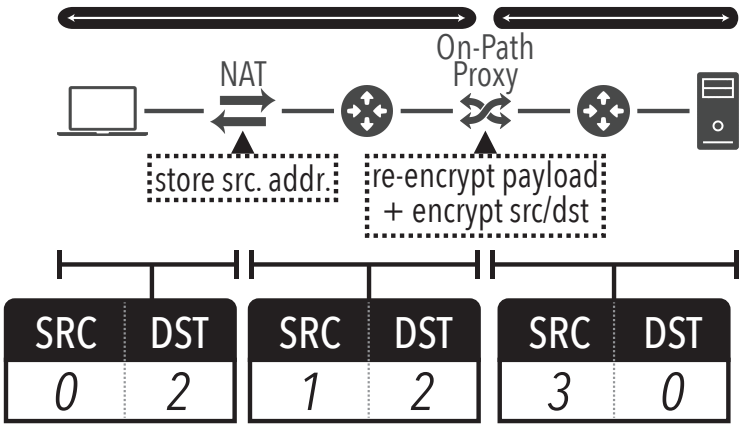


Figure 6.7: **Evaluating a Path.** Dashed boxes mark privacy which primitives a device performs. The numbers below the path indicate the effective sender and receiver radii leaked by packet headers in that region of the path. The bars above the path indicate segments of the path in which sightings from multiple vantage points can be linked.

full network topology, and no traffic matrix, just a path of devices. The procedure, illustrated in Figure 6.7, is as follows:

1. Mark which primitives each device implements, if any. *(Represented by dashed boxes in the figure.)*
2. Mark boundaries of linkable segments. We consider devices that (1) alter the appearance of the payload (e.g., by re-encrypting it) and (2) change header fields that uniquely identify the flow (e.g., the standard 5-tuple) to break linkability. Additionally introducing artificial delay offers stronger guarantees against timing attacks, but adds so much overhead we do not require it. *(In the figure, the proxy re-encrypts packets, so it interrupts linkability.)*
3. Mark boundaries of regions where headers reveal different information (i.e., devices that encrypt, store, or omit state). *(In the figure, the NAT stores the packet's source address and replaces it with the network's address. The receiver's address is encrypted on the sender side of the proxy and the sender's address is encrypted on the receiver side of the proxy.)*
4. Assign 4-tuple *(Sender Radius, Receiver Radius, Sender Radius Reduction, Receiver Radius Reduction)* to each region with a vantage point. Regions where indirection is used to send the packet off-path have sender and receiver radii of ∞. *(Assume each link in the example is a vantage point. For clarity, the figure only shows effective radii, not radius reduction.)*
5. Merge the radius tuples for regions in the same linkable segment by taking minimum effective radius and the maximum radius reduction across regions in a segment. *(In the figure, the first linkable segment has minimum (sender, receiver) radii of* $(0, 2)$*. The second segment already has only one region,* $(3, 0)$*.)*
6. Report (1) a list of the 4-tuples for the linkable segments *(in the example:* $[(0, 2), (3, 0)]$*)* and (2) a list of costs associated with the primitives marked in step 1 *([1x Storage, 1x Key Dist, 2x CPU])*.

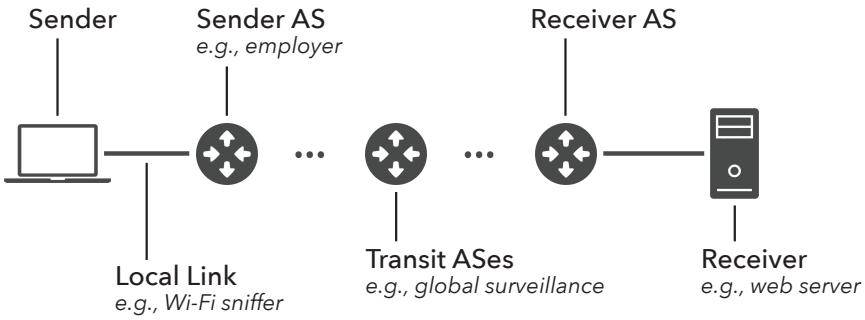


Figure 6.8: **Canonical Path.** In the absence of concrete topologies, we evaluate using a path that includes devices representing common real-world adversaries plus any architecture-specific devices, like NATs or Tor relays. Dots represent a number of unspecified hops.

Evaluating without a Path. Now suppose we want to evaluate a protocol but we do not even have a concrete path from sender to receiver (and, of course, still no traffic matrix)—in general, this is the case when we evaluate a protocol at the highest specification level (Section 6.2). The process will be very similar to that presented above in the case *with* a concrete path, except instead of hard hop numbers, we will use variables to represent the length of certain parts of the hypothetical path.

In the absence of a concrete path, we define a *canonical path*—a generic sequence of devices that capture the protocol's behavior and most common adversaries. Shown in Figure 6.8, the canonical path consists of: (1) the sender and the receiver; (2) three routers, representing the sender's AS, the receiver's AS, and transit ASes; and (3) any devices relevant to the privacy protocol being evaluated (e.g., a NAT or a Tor relay). The dots in the figure represent an unknown number of unspecified hops. The first two pieces (sender, receiver, and routers) ensure that the path captures several real-world adversaries:

- **Local Eavesdropper** *(e.g., someone sniffing open Wi-Fi):* Put a vantage point on the link leaving the sender.
- **Global Surveillance** *(e.g., monitoring by a nation-state):* Put a vantage point on the "transit ASes" router.
- **Source ISP** *(e.g., employer):* Put a vantage point on the "source AS" router.
- **Receiver** *(e.g., web server):* Put a vantage point on the receiver.

The third piece, protocol-specific devices, is needed because devices like Tor relays affect privacy by operating on packet contents; modelling them also allows us to represent privacy breaches if they are compromised. In a real deployment, paths may contain *more* devices, but we know every path will contain *at least* these devices.

Now, let $h(X, Y)$, abbreviated h_Y^X, be a function that gives the number of hops to X from Y (Figure 6.9). We will use the notation $h^X = h(X, \cdot)$ to mean the distance to X from an unspecified Y. We can now describe the distance to the sender from any point on the path as h^S and the distance to the receiver as h^R; this notation is useful to describe the radius observed along a region of the path consisting of multiple vantage points. For example, in Figure 6.10, the sender radius reduction before the NAT is $h^S - 1$ (that is, the headers give away all the remaining hops to the sender) whereas the sender radius reduction after the NAT is $h^S - 2$ (the headers only give away the path up until the NAT). Note that the effective radius and radius reduction values always sum to

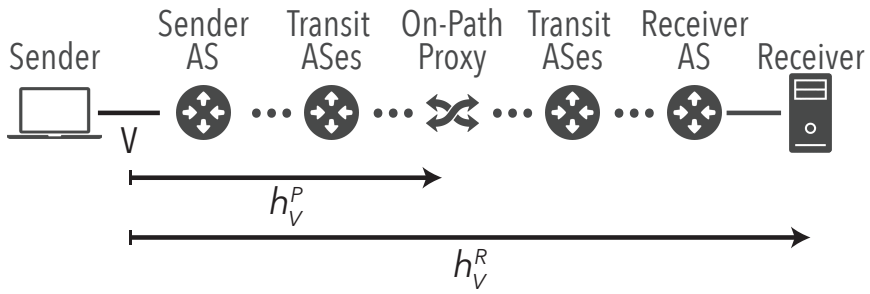


Figure 6.9: **Variable Hop Counts.** Without a concrete path, the hop count from X to Y is represented as $h(X, Y)$, or h_Y^X. In the figure, the distance from the vantage point V to the receiver is h_V^R hops; the distance to the proxy is h_V^P hops.

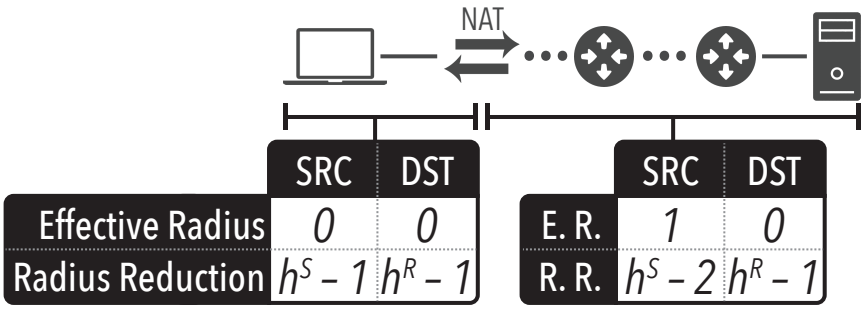


Figure 6.10: **Variable Radius Reduction.** The notation h^S or h^R can be used to describe the number of hops to the sender or receiver from any vantage point in a region of the path. For example, the sender radius reduction for any vantage point to the left of the NAT is $h^S - 2$.

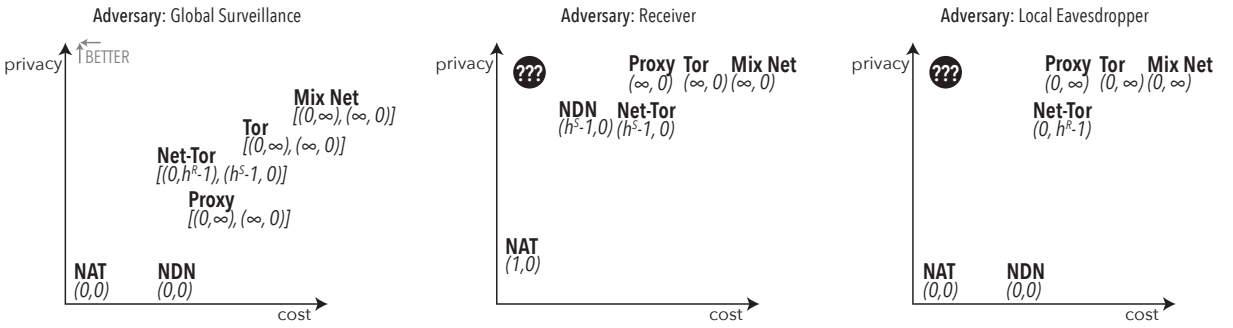


Figure 6.11: **Privacy vs. Cost Design Space.** These plots compare architectures in terms of privacy and cost with respect to three different adversaries: global surveillance (left), the receiver (center), and a local eavesdropper (right). In each case, an ideal architecture (top left corner) is conspicuously absent.

$h^{\{S,R\}} - 1$ because the baseline radius is $h^{\{S,R\}} - 1$ and the radius reduction is the difference between the baseline and effective radius.

6.4.2 Evaluating Existing Architectures

Table 6.4 lists some network architectures from practice and from research. The second column describes which privacy primitives each one uses, the effective radii for a global observer, and the total cost of the primitives. The third column shows privacy diagrams like the ones introduced in Section 6.4.1.

"IP" and "IP with NAT" are straightforward (but note that we do not consider IP multicast); "Overlay Tor" refers to the standard Tor system with three overlay onion relays in the path. "Network-Layer Tor (Net-Tor)" is a recently proposed system where layer 3 routers act as onion relays [157, 158]; in our examples, we assume that the onions only protect header information hop-by-hop and that the packet takes the normal default path to the receiver (i.e., there is no path stretch) and that every router in the path participates (not just 3). "Proxy" is a single intermediary server that swaps headers and re-encrypts the payload, like a single Tor relay. This also captures the basic behavior of i3 [217]. NDN [139] is a content-centric network architecture whose salient feature here is its lack of source addresses: every router remembers on which input port a request for a given piece of content arrives; responses follow this path of "breadcrumbs" back to the original requester. Destination addresses, which name pieces of content rather than hosts, are fully visible along the whole path. (Because of the change in destination address semantics, the meaning of the receiver anonymity radius is somewhat different for NDN. In IP, the destination address makes it clear *who* the sender is speaking to, which, in the case of the Web at least, can often be mapped to *what* content they are retrieving. In NDN, this mapping is no longer needed for the adversary to determine *what*, but determining *who* may now require vantage points close to content sources.) Finally, though not pictured, a "mix network" [75] is essentially Tor with artificial delay and re-ordering at each relay.

Figure 6.11 places these architectures in relation to one another in terms of privacy and cost with respect to three different adversaries: global surveillance, the receiver, and a local eavesdropper (Section 6.4.1). Privacy rankings are based on each architecture's (src, dst) radius score. Cost rankings are somewhat subjective; directly comparing different kinds of overhead is difficult—

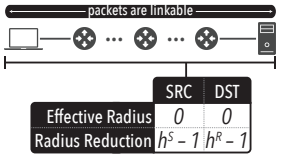
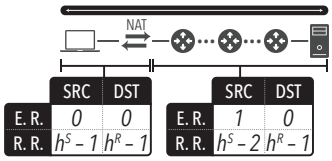
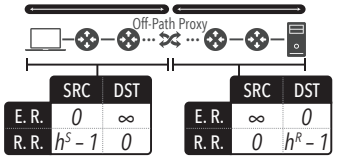
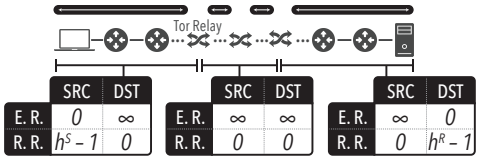
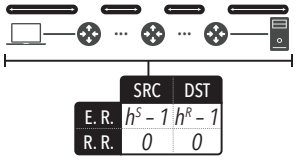
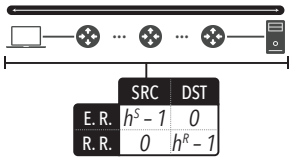
Protocol	Primitives & Cost	Privacy Diagram
IP	**Source:** None **Destination:** None **Linkability:** None **Cost:** [ ]	
IP w/ NAT	**Source:** Distribute State **Destination:** None **Linkability:** None **Cost:** [1x Storage]	
Proxy	**Source:** Encrypt & Distribute State, Indirection **Destination:** Encrypt & Distribute State, Indirection **Linkability:** Scramble State **Cost:** [1x Bandwidth, 1x Latency, 1x Key Dist, 2x CPU]	
(Overlay) Tor	**Source:** Encrypt & Distribute State, Indirection **Destination:** Encrypt & Distribute State, Indirection **Linkability:** Scramble State **Cost:** [3x Bandwidth, 3x Latency, 3x Key Dist, 6x CPU]	
Network-Layer Tor	**Source:** Encrypt & Distribute State **Destination:** Encrypt & Distribute State **Linkability:** Scramble State **Cost:** [h_R^Sx Bandwidth, h_R^Sx Key Dist, h_R^Sx CPU]	
NDN	**Source:** Distribute State **Destination:** None **Linkability:** None **Cost:** [h_R^Sx Storage]	

Table 6.4: **Privacy Protocols.** Some architectures from practice and from research, the privacy primitives they use to hide communication state, and their privacy properties. Each diagram shows the canonical evaluation path, the effective radii and radius reduction scores at different locations along the path (below), and the linkable segments (above).

which one is "worse" depends on personal priorities and whether you ask a user or a network operator. For example, in both NDN and Net-Tor each router implements a primitive, but in the former case routers store state and in the latter case they perform cryptographic operations.

Two high-level takeaways emerge at a glance. First, an architecture's placement changes from adversary to adversary because vantage point location impacts effectiveness (leakage is not consistent across the path). A particularly striking example of this is NDN: a vantage point on the first-hop link from the sender has a score of $(0, 0)$ whereas the last link before the receiver has a score of $(h_R^S, 0)$—that is, a vantage point near the sender is very serious in NDN whereas a vantage point near the receiver (or the receiver itself) is harmless. Second, none of the plots has an architecture in the (ideal) top left position! We return to this point in Section 6.5.

Global Adversary *(left)***.** First, NATs and NDN do nothing against a global adversary because in both cases a vantage point on the link leaving the sender learns the sender from topology and learns the receiver because its address is on display in the headers; both systems earn a privacy score of $[(0, 0)]$. Next, note that mix nets, Tor, and proxy have identical privacy scores: $[(0, \infty), (\infty, 0)]$. We place them at different levels of privacy, however, due to subtle differences in linkability: mix nets use artificial delay to make linking sightings difficult; Tor is more susceptible to timing attacks. A proxy is like Tor but with only one relay, so linking sightings based on timing is easier yet. Finally, we rank Net-Tor slightly above proxy in terms of privacy because, even though its radius score is slightly worse—the adversary learns 1 hop instead of 0—linking sightings across the path requires more work since each router acts as an onion relay (compared to the single proxy).

Receiver *(center)***.** First, the differences in privacy ranking between mixes, Tor, and proxy have disappeared. In the case of the global adversary, that differentiation was caused by differences in the adversary's ability to link sightings from different vantage points. In this case, the adversary has just a single vantage point (the receiver). Second, NDN is suddenly very effective, and at a modest cost. NDN is so effective because the last-hop router before the receiver implements a privacy primitive; this gives the biggest sender radius possible without adding any indirection latency since the router was on-path already. No anonymity system today can provide last-hop functionality for every possible receiver; they all involve overlay nodes multiple hops away. (More on this in Section 6.5.) Finally, against the receiver, a simple NAT helps slightly; preventing the adversary from identifying which host in the source network is the sender may be useful.

Local Eavesdropper *(right)***.** The results here are similar to the receiver case, except NAT and NDN drop back to $(0, 0)$. Given that a local adversary is relatively weak (picture a Wi-Fi sniffer in a coffee shop), it is particularly apparent—and frustrating—that no strong, low-cost solution exists.

6.5 A Build-Your-Own Privacy Service

We saw in Section 6.4.2 that there are some surprising holes in the privacy vs. cost design space. Specifically, it is surprising that there is no highly private yet low cost solution for relatively weak

Primitive	Pricing Guidelines
Indirection	Charge per extra link due to extra traffic. (The corresponding latency increase also impacts user experience, but not pricing.)
Artificial Delay	Charge per router due to need for increased buffer space. (Again, latency also hurts UX.)
Dummy Traffic	Charge per kB per link (for extra traffic).
Multicast	Charge per kB per extra link (extra traffic).
Encrypt State	Charge per router (for extra computation). (Header overhead handled implicitly with existing data cap.)
Distribute State	Charge per router (for per-flow storage). (Fate sharing also lowers reliability for user.)
Scramble State	Charge per router (for extra computation).

Table 6.5: **Pricing.** ISP pricing guidelines for each primitive.

adversaries like the receiver or a local eavesdropper. More generally, is it possible to give users exactly the privacy/cost tradeoff they want for each connection? Today we have a small number of anonymity systems that cover just a few discrete points in the design space; what if we could create exactly the protocol we needed, at any point in the design space, on an on-demand basis?

Imagine ISPs wanted to help solve this problem by offering anonymity as a paid service. This service would let users customize their protection to guard against a particular adversary but without paying for unnecessary protection. To do this, ISPs could provide a "build-your-own-protocol" service by offering the privacy primitives in Table 6.1 on a per-router (or per-AS) basis. The user (with the help of some configuration tool) would essentially build a custom protocol to address their needs by enabling certain primitives on certain routers. ISPs assign a cost to each primitive and charge users based on which primitives they enable, on how many routers, and how much traffic they send through that configuration (see Table 6.5). (For the moment, we blithely ignore practical issues like user interfaces, limited user expertise, and coordination across ISPs.)

Example Scenarios. To motivate the usefulness of such a service, we describe how it could help in the case of three common adversaries users worry about for which today's privacy systems are not a perfect fit.

Server Logs. Users may want to hide their identify at the receiver, either to be anonymous from the service itself, or because logs might be subpoenaed or stolen. This is a very simple adversary—there is only one vantage point, so linkability never even enters the picture. All that's needed is to use the last-hop router before the receiver to remove any information identifying the sender from the packet. This can be done by either storing the sender's address on the router and replacing it with the router's address in the packet ("Store Previous Hop"), or by giving the receiver a stack of headers $(Receiver \to Router, \{Router \to Sender\}_{K_{Router}})$ to respond with, where the $Router \to Sender$ header is encrypted for the router ("Carry Previous Hop"). In either case, the radius score at the receiver is $(h^S - 1, 0)$ (Figure 6.13).

Public Wi-Fi Sniffer. Users using public Wi-Fi at airports or coffee shops may want to hide their

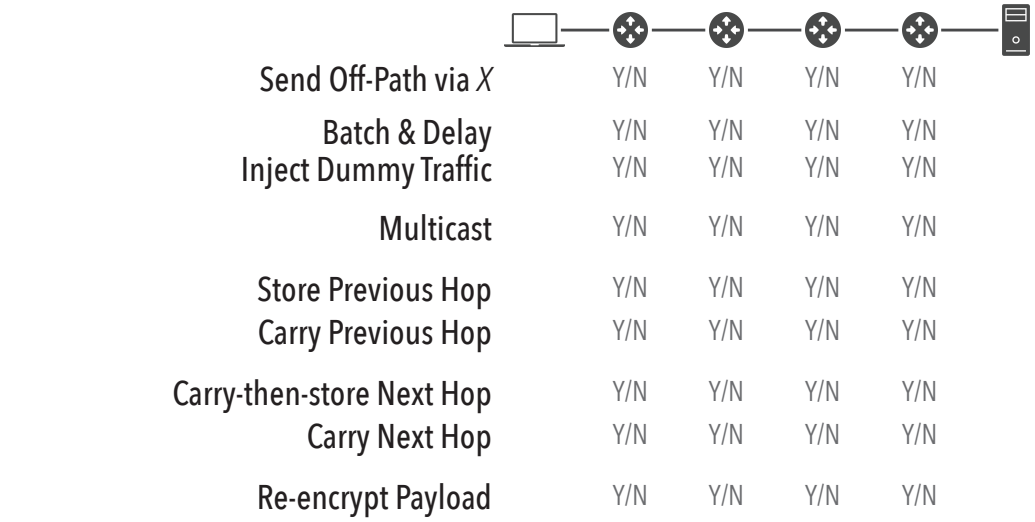


Figure 6.12: **Building a Protocol.** Users can customize their level of protection by enabling primitives on specific routers.

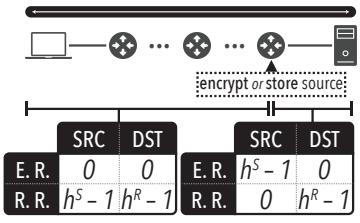


Figure 6.13: **Scenario: Server Logs.** Hiding the sender's identity from the receiver only requires help from the last-hop router.

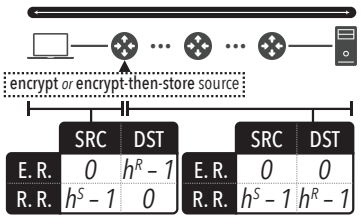


Figure 6.14: **Scenario: Public Wi-Fi Sniffer.** Hiding the receiver's identity from local eavesdroppers only requires help from the first-hop router.

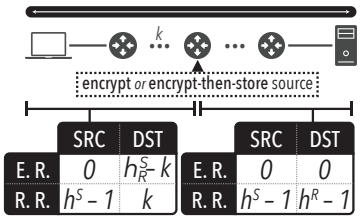


Figure 6.15: **Scenario: Untrusted ISP(s).** Hiding the receiver's identity from an access network that controls the first k hops only requires help from router $k + 1$.

activity from local eavesdroppers. This is the mirror image of the previous example: simply include an encrypted next-header for the first-hop router (Figure 6.14). This could be done for every packet ("Carry Next Hop"), or the router could store this information for subsequent packets ("Carry-then-store Next Hop").

Untrusted ISP. There may be cases where users do not trust their access network (e.g., they may be travelling and accessing the Internet from a country that practices censorship or surveillance). This is similar to the previous scenario, except now the adversary might have vantage points on the first k links. By using one of the primitives described above on router $k+1$, the adversary's receiver radius is $h_R^S - k$ (Figure 6.15).

These examples show why a system like the one proposed in this section could shine: In all three scenarios, which consider adversaries located either close to the sender or close to the receiver, there is no existing solution in the top left corner of the privacy vs. cost plots in Figure 6.11 (center and right). The primary reason for this is that there is no way to place primitives near the adversary (where they are most effective) and only there (to keep costs down). Anonymity systems today, like VPNs or Tor, are forced to operate as overlays, since not all routers are willing to participate, which limits how close to the adversary we can place primitives. This limits effectiveness. And proposals like NDN and Net-Tor, which do push functionality to the routers themselves, do not allow precise control over *which* routers participate, increasing cost unnecessarily.

Custom Protection. In general, to build a custom anonymity protocol for a particular adversary (set of suspected vantage points), the user needs to decide which primitives to enable at each router (picture enabling switches in the grid in Figure 6.12).

(1) Limit identity leakage at each vantage point. Since by default each vantage point can see 1 hop forward and 1 hop backward (the T&T baseline), we start by hiding forwarding state on the routers immediately up- and downstream of each vantage point. Sender identity can be hidden using multicast, encryption ("Carry Previous Hop"), or in-network state ("Store Previous Hop"). Destination state is similar, with one important difference. In the case of the source, a router just needs to know the previous hop, which it already knows without extra information in the header because it saw where the packet came from. For the destination, on the other hand, a router needs to be told the next hop, which requires some kind of information from the packet itself. So, next-hop information can either be carried (encrypted) in every packet ("Carry Next Hop"), or be carried in the first packet, stored on the router, and omitted in subsequent packets ("Carry-then-store Next Hop").

Now the only leakage is due to the T&T baseline. If a vantage point is near one endpoint, the baseline gives away a lot of information about that endpoint and very little about the other; based on this, the adversary is unlikely to link the sender and receiver (Figure 6.16 top). On the other hand, if a vantage point is near the middle of the path, the baseline gives away a moderate amount of information about both the sender and receiver (bottom). If this is unacceptable, it may be necessary to enable indirection.

(2) If there are multiple vantage points, break linkability between consecutive vantage points. Simply re-encrypting the packet at an intermediate router might suffice, but, for a higher cost, enabling artificial delay as well will decrease the chances the adversary can link packets.

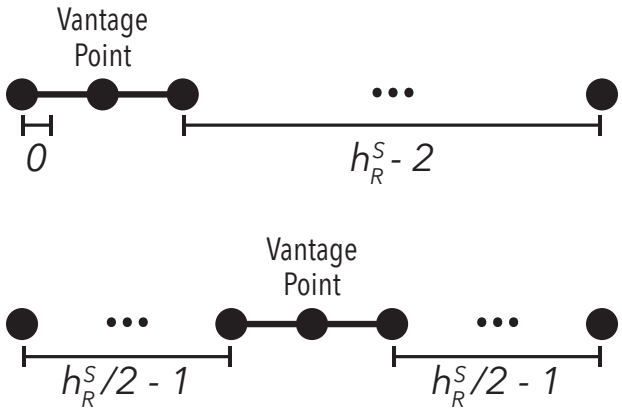
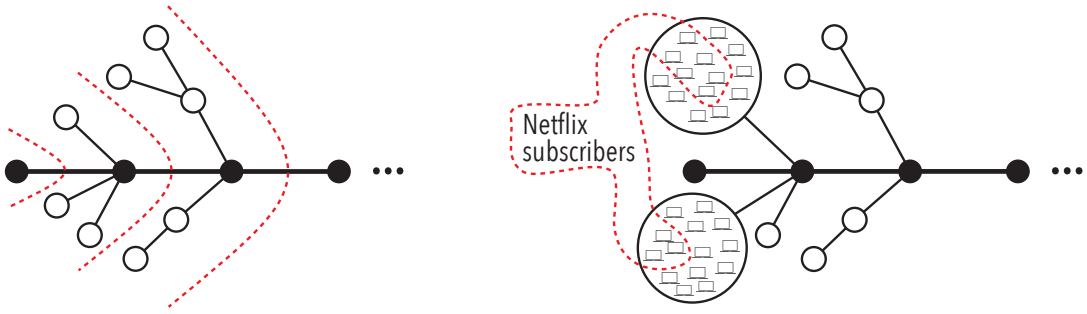


Figure 6.16: **Impact of Vantage Point Location.** A vantage point near one endpoint learns a lot about that endpoint and very little about the other, whereas a vantage point in the middle of the path learns a moderate amount about both endpoints.



(a) Anonymity Radius.

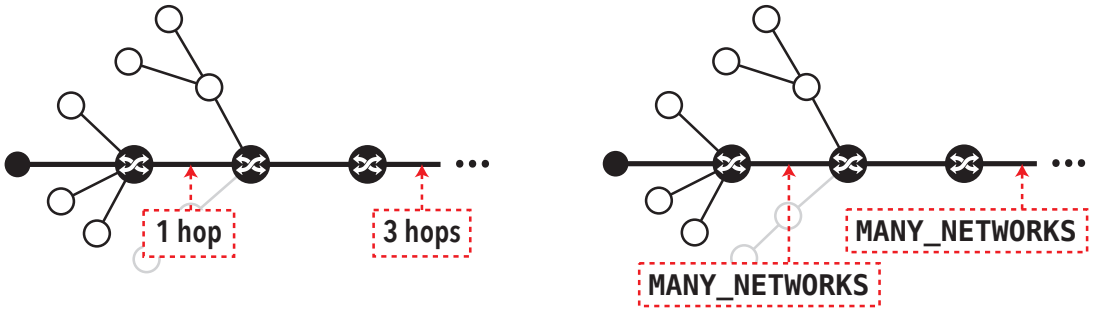
(b) Share counts.

Figure 6.17: **Anonymity Radius vs. Share Counts: Anonymity Set Shape.** Anonymity radius can only express anonymity sets that are subsets of the network topology, as shown by the dashed lines (left). Share counts can incorporate external information to express anonymity sets like "users in networks A and C who are also Netflix subscribers," again shown by a dashed line (right).

6.6 Conclusion

This chapter explores privacy at the network layer. We offer structure to the discussion by modelling the problem—the adversary workflow—and by articulating the basic building blocks underlying most anonymity systems—the privacy primitives. Then we address the problem of evaluating a network architecture in terms of privacy. Unlike previous approaches, ours does not require specific information about a concrete *deployment* of an architecture, but rather evaluates the intrinsic privacy properties of the *architecture itself*. We use our methodology to evaluate existing anonymity systems and to propose a new "build-your-own" privacy service.

Share Counts vs. Anonymity Radius. In this chapter, we proposed two metrics, share counts and anonymity radii ("hop counts"). Both are abstractions for describing the size of an anonymity set.



(a) Anonymity Radius. (b) Share counts.

Figure 6.18: **Anonymity Radius vs. Share Counts: Anonymity Set Size.** Without a concrete topology, anonymity radius can roughly quantify how many networks are in the anonymity set (left), whereas share counts cannot (right). This example shows network-layer Tor.

Hop counts, however, can only express anonymity sets based on topology-related identity information (Figure 6.17a). That is, they work by partitioning off a subset of the (hypothetical) network topology as the anonymity set and their only degree of freedom is selecting the size of that subnet. Share counts, on the other hand, are much more general; they can express anonymity sets that do not neatly follow topological boundaries (Figure 6.17b). For example, suppose an attacker has access to external information like a list of subscribers to a particular Web site (see Table 6.2 for more examples). If it combines this with knowledge of a packet's source and destination networks, it can produce an anonymity set that is the intersection of hosts in the source network and subscribers to the destination service. Share counts can capture this, and so in a sense have the potential to be more precise than hop counts.

Additionally, share counts can capture privacy-related properties beyond sender and receiver anonymity sets. First, the sender flow share count, which indicates how many of one sender's flows share the same Sender-ID, captures how much of one user's activity the adversary can link together. The anonymity radius metric has no analog for this. Second, the Flow-ID share count quantifies how much collateral damage will result from a flow being blocked (e.g., in AIP or APIP). Finally, if share count analysis were extended to run over multiple (real or hypothetical) paths with a consistent set of (real or hypothetical) hosts, the Sender-ID share count would expose the fact that in architectures with persistent host IDs, like AIP or XIA, a host's activity could be tracked across multiple networks (e.g., a campus network and public Wi-Fi in a coffee shop).

On the other hand, in the absence of concrete information like a network topology or external information, share counts have less discerning power than hop counts. Without a concrete topology, share counts use the constants `ONE_HOST`, `ONE_NETWORK`, `MANY_NETWORKS`, and `WHOLE_INTERNET` in place of actual numbers. Share counts lack a way to distinguish, for example, in an architecture based on network-layer Tor, what an adversary learns from a vantage point two hops from the source versus five hops from the source (in each case, the anonymity set size is `MANY_NETWORKS`) (Figure 6.18b). Hop counts were designed specifically to disambiguate these cases without concrete topologies (Figure 6.18a).

Part IV

Conclusions and Future Work

Chapter 7 Conclusion

This thesis describes the motivation, design, and evaluation of network architectures and protocols that, taken together, deliver on our overarching vision: a network that protects users' privacy without compromising on performance, accountability, functionality, or security.

This is surprisingly hard to achieve, and the underlying issue is controlling access to information. In the case of data privacy, that information is the application layer data carried in packet payloads; in the case of metadata privacy, that information is the network source address. In each case, there are both legitimate uses and malicious uses of this information in the network. We have discussed these at length in this thesis: given access to application data, the network can serve requests from caches, compress data, or detect attacks; given access to the source address, operators can find attackers. At the same time, users would sometimes like to hide this information from the network to prevent malicious uses, like identity theft, harassment, or imprisonment by a hostile government.

The current Internet architecture was not designed to hide any information from the network, so it enjoys all of the performance, accountability, functionality, and security benefits described above but provides no privacy. On the other hand, protocols like Tor and TLS go to the other extreme: Tor hides network headers and topology information and TLS hides application data, providing privacy but precluding any beneficial uses of that information. This thesis argues that if we carefully control which entities can access or alter each piece of a packet, and potentially what they can do with that information when they do, we can achieve a balance of all of these properties. In order to do this, however, support from the network is required. This is because the network uses the information in question, so unilaterally changing where it is stored (e.g., changing the meaning of header fields) or how it is stored (e.g., encrypting what used to be plaintext) would disrupt the operation of the network.

7.1 Contributions

In Part II, we explored controlling access to packet payloads. As we saw, the payload was traditionally accessible to *anyone*, so an ecosystem of middleboxes sprang up to perform optimizations like caching or security functions like virus scanning. On the other hand, payloads carry sensitive user data, so common practice today is to use encryption so that *only endpoints* can access it. In this part, we argue that "anyone" is too permissive and "only endpoints" is too restrictive and introduce two new protocols, mcTLS and mbTLS, as intermediate points.

Chapter 2 presented our measurement study on the impact of using TLS, which showed that the loss of middleboxes is the most serious negative effect of switching to HTTPS. Other potential

costs were either already minimal (e.g., increased computation or energy consumption) or can be "engineered away" with protocol improvements (e.g., QUIC [124], Zero [137], and TLS 1.3 [196] are all working toward eliminating TLS's extra round trips).

Chapter 3 introduced the idea of *encryption contexts*, an abstraction endpoints can use to restrict which parts of the data stream each middlebox can access and whether that access is read/write or read-only. We presented Multi-Context TLS, a protocol that implements the encryption context abstraction in TLS. mcTLS applies the *principle of least privilege* to middleboxes by allowing endpoints to give them the minimum amount of access they need to do their jobs.

Chapter 4 took a broader perspective and described a design space for secure multi-entity communication protocols, which it used to put previous proposals (including mcTLS) into context. Next we identified gaps in this design space where real-world needs were not satisfied and presented Middlebox TLS, a protocol offering techniques for discovering middleboxes on-the-fly, protecting middlebox path integrity, and securing middleboxes that are outsourced to third party hardware.

Though we have focused on the technical insights and implementation artifacts, perhaps the most important impact of mcTLS and mbTLS is encouraging discussion of encrypted traffic interception. The prevailing opinion in some large technology companies and standards bodies is that, because application data could be used maliciously, it should be encrypted end-to-end, period. In our view, this argument (1) dismisses the benefits of middleboxes without carefully weighing them against privacy benefits of end-to-end encryption and (2) glosses over the reality that, as we have discussed, TLS interception via mechanisms that are not secure (e.g., custom root certificates) is commonplace today and unlikely to disappear in practice. In this light, we see mcTLS and mbTLS as *strengthening* privacy, not weakening it—saying that these protocols weaken privacy compared to end-to-end encryption is true, but end-to-end encryption is the wrong baseline. A conversation about the risks and benefits of encrypted traffic inspection is crucial, and we hope to see it continue.

In Part III, we turned our attention to controlling access to packet source addresses. Here, we were concerned with both who can see them (privacy) and how trustworthy they are (accountability). We saw that the current Internet does a poor job on both accounts: source addresses are visible to everyone and are easily spoofed. Previous work fixes only one problem or the other—for example, AIP [40] makes source addresses unspoofable but leaves them visible to *anyone* whereas Tor [96, 158] makes a packet's true source address visible to only *one Tor relay*. Once again, in this part, we establish a practical middle ground.

Chapter 5 presented a new network architecture, the Accountable and Private Internet Protocol (APIP), that explicitly balances privacy and accountability—and, in fact, offers more of each than the current Internet, without introducing significant overhead in the default case. We argued that the reason for the tension between accountability and privacy is that the IP source address is overloaded: it serves both as a "return address" (identifying the packet's sender) and an "accountability address" (acting as a handle for finding malicious senders). APIP recognizes this and explicitly separates the two, giving each packet a destination address (as before), an accountability address, and a return address.

Chapter 6 answered the question: how can we measure "how private" a network architecture is? We presented two new metrics, share counts and the anonymity set radius, for quantifying an architecture's privacy properties even in the absence of a concrete deployment (topology and traffic).

A good guiding principle in research is to question assumptions. The insight behind APIP came from revisiting basic assumptions about network source addresses. In Section 5.2 we identified five roles currently fulfilled by source addresses in IP. By disentangling two of them—return address and accountability—we were able to balance privacy and accountability. This experience suggests it may be worth revisiting the remaining roles and considering whether conflating them is causing other problems. For instance, could anything be gained by explicitly separating error reporting and return addresses? What about other header fields—are there other basic assumptions worth revisiting?

7.2 Impact

- **Academic Recognition:**
  - APIP [176], from Chapter 5, received a Best Paper Award at *SIGCOMM 2014*.
- **Industry Recognition:**
  - mcTLS [177], from Chapter 3, has been patented by Telefónica.
  - mcTLS is in the standardization process at the European Telecommunications Standards Institute (ETSI) [34].
  - mbTLS, from Chapter 4, has been patented by Microsoft.
- **Open Source:**
  - Our library for Web page load experiments, developed for Chapter 2, is available on GitHub [30].
  - mcTLS, from Chapter 3, is available on GitHub [3].
- **Education:**
  - mcTLS, from Chapter 3, has been included in courses such as "Understanding and Securing TLS" (CS 6501) at the University of Virginia and "Graduate Level Security" (CMSC 8180) at the University of Maryland.
  - APIP, from Chapter 5, has been included in courses such as "Computer Networks" (15-744) at Carnegie Mellon, "Network Security" (18-731) at Carnegie Mellon, "Advanced Computer Networks" (CS 538) at UIUC, "Security and Privacy" (ELE 574) at Princeton, and "Computer Communications" (CS 5220) at the University of Colorado.
- **Other:**
  - mcTLS [177], from Chapter 3, was featured on *The Morning Paper* [83], a popular blog that posts "an interesting/influential/important paper from the world of CS every weekday morning."

Chapter 8 Future Work

8.1 Near-Term

Formal Protocol Verification. We have not formally verified mcTLS or mbTLS. Instead, in the security analyses presented in Section 4.4 and Appendix B, we assume TLS is a secure starting point, make as few changes to TLS as possible, anticipate possible attacks against the changes we do make, and argue why those particular attacks cannot happen. Even features that appear simple, however, can interact in unexpected ways that can undermine security; formal verification is important for catching these cases. In fact, TLS itself has not yet been formally verified in its entirety, though recent work is making steady progress toward this [56, 58, 60, 61].

From Trusted Computing to Trusted Systems. With mbTLS, we have so far focused on the core protocol design and middlebox software architecture; we have largely ignored practical issues even though they are equally important to mbTLS's real-world success.

One class of issues is "implementation details" like key and identity management. There are many problems to solve here: how do we handle sharing middlebox state across replicated instances when sessions are bound to SGX enclaves on a particular CPU? And since mbTLS uses remote attestations to prove that a particular piece of software is running on the middlebox, how do endpoints know what software identity to expect? And how can we manage keys to prevent Cuckoo attacks, where a powerful adversary invests resources in cracking a single SGX CPU in order to forge attestations in the future?

Another class of problems pertains to designing middlebox software that runs on trusted hardware—it is more complicated than "just drop the code as-is into an enclave." First, we need analysis tools that can track the flow of sensitive information to make sure it cannot inadvertently leak outside the enclave (implicitly or explicitly). This includes understanding when a program is vulnerable to side channel attacks (e.g., cache timing attacks). Second, is it important to isolate session state from different clients? If so, can we, e.g., spin up a new enclave for each connection, or at least each user?

Making APIP Practical. Chapter 5 presents the high-level design of APIP plus a first-pass feasibility evaluation based on one packet trace from a university network. To show how to make APIP deployable in real life, there are a number of low-level details to work out (APNA already tackles many of these details [155]). For example:

Delegate Scalability Clients need to send packet fingerprints to their delegates in a way that does not

overwhelm the network, storage on the delegate, or the delegate's ability to search for a fingerprint when it receives a **verify()**. There are many parameters to tune here: for example, assuming clients store fingerprints of outbound packets in Bloom filters, how often should they send those Bloom filters to their delegates? More often means more filters for the delegate to store and search, but shorter verification latencies since delegates get briefs sooner. Another example is flow ID granularity. If more hosts share the same flow IDs, they get better privacy (two flows with the same ID did not necessarily come from the same sender). But, since delegates search for briefs based on flow ID, more clients sharing a flow ID means more Bloom filters to check for each **verify()**.

Flow Setup Latency If implemented naively, waiting for a **brief()** and one or more **verify()**s at the start of each TCP flow will cause serious performance problems for applications that use many short-lived connections (e.g., like web browsing). There are various solutions to explore here: verifiers could rate limit or allow N packets from each flow to go through while they wait for responses from delegates. This requires routers to keep state, however; another possibility is letting TCP SYN or TLS HELLO packets to pass through unverified, though this may not be feasible for security reasons. If we expect multiple routers on a path to perform verification, perhaps it will be necessary to "staple" verifications along with the first packet of each flow to avoid a round trip to the delegate at each verifier.

A Deeper Analysis of Architectural Privacy. We see the privacy metrics presented in Chapter 6 as initial steps. Currently, they serve largely to structure a manual analysis of an architecture or protocol—a sort of mental checklist. Most of the process requires a human expert with a thorough understanding of how an architecture's network elements interact with packets and what these interactions mean for privacy. These metrics could be much more useful, however, if combined with techniques for automatically discovering how protocol features interact to impact anonymity with minimal human guidance. Given a protocol spec, a packet trace, or even source code, could a tool learn the mappings from header fields to meta-fields that we use in share count analysis? For example, based on packet traces, could such a tool deduce on its own that the address/protocol/port 5-tuple serves as a flow ID in TCP/IP? Or that TCP sequence numbers can be used to link packets in the same flow with high probability?

8.2 Long-Term

A Full-Stack Approach to Privacy. This thesis focuses on privacy at the network and transport layers. But the other layers cannot be ignored: MAC addresses can be used to track users, and application layer data contains all kinds of personally identifiable information. Privacy at these layers has seen plenty of attention (e.g., [11, 13, 108, 121, 161, 184, 222]), but this is not enough. Prior research argues that a true privacy solution must consider all layers of the network stack. To effectively switch identities, users must use a new identifier at each layer; this set of identities is called a *pseudonym* [126]. For example, at the very least, a pseudonym would consist of a new MAC address, a new IP address, and a new collection of cookies in the browser.

Efforts in this vein are related to multiple parts of this thesis. For example, mcTLS could be used to hide from middleboxes information that could be used to uniquely identify a user, like the

user agent HTTP header. Share counts could be generalized across layers—for instance, based on traffic traces or server logs, users could be told how common (or uncommon) requests from their browsers look (e.g., are you identifiable by your HTTP headers?). In APIP, a cross-layer pseudonym would also include a new flow ID. One common strategy might be to use a different pseudonym for each website a user visits. Based on this and the other strategies proposed by Han et al. [126], it would be interesting to calculate how many flow IDs each delegate would need and how it could recycle flow IDs across its clients.

A Verifiable Internet: Bringing Trusted Computing to the Network. Currently, many properties of our network communications are difficult for us to verify; we must blindly trust our ISPs. For instance, if an ISP claims to be net neutral—that it does not provide packets from different senders or applications with different service qualities—we are currently forced to verify this by reverse-engineering its traffic shaping policies through external measurements. Another example is privacy: we know intelligence agencies work with major ISPs to track end-to-end communication patterns, but we do not know exactly what information ISPs release to authorities and under what conditions.

By implementing pieces of router functionality in trusted environments like SGX enclaves, we could start to change this "network-as-a-black-box" dynamic. Endhosts could verify the identity of router software/firmware/hardware, thereby also verifying properties about its behavior. Furthermore, if one of the behaviors of the verified software is to perform the same verification on all neighboring routers, a host connecting to the network only needs to verify its first-hop router in order to have guarantees about the behavior of, ideally, the entire Internet. (This idea was used in TrueNet to localize malicious or faulty routers [241].)

Consider a specific example: just like Web services have privacy policies about the information they collect, so too could ISPs. For example, an ISP might promise to only disclose logs of source/destination IP address pairs in response to subpoenas signed with a certain authority's private key. Using remote attestation, an endhost (or an edge network on behalf of its customers) could verify that all routers on a packet's path run software that abides by these privacy policies.

As a starting point, rather than hosts verifying correct behavior of the network, the network could verify correct behavior of the hosts [186], ensuring, for example, that hosts do not spoof source addresses, that they honor source quench messages (a DoS defense mechanism), and that they implement congestion control honestly (rather than greedily taking more than their fair share when the network is congested). Ideas like these were floated in the early days of trusted computing (e.g., the Trusted Computing Group's "Trusted Network Connect"), but recent technological advances like SGX now put them within practical reach.

Appendices

Appendix A mcTLS Protocol Details

This documentation, with primary contributions from Kyle Schomp, is included here for completeness. It is also available at [1].

The mcTLS protocol is designed to enable secure communication between a client, server, and 1 or more middleboxes in series between the client and server. The protocol is built as an extension to the TLSv1.2 protocol. This document extends and modifieds the TLSv1.2 specification [94]. Where details are omitted here, the reader should assume that no changes are made from the TLSv1.2 specification.

A.1 Goals

mcTLS is designed to provide the following five properties:

1. **Entity Authentication:** As in TLS, the client must be able to verify the identity of the server. Additionally, the endpoints must be able to authenticate each middlebox.
2. **Payload Secrecy:** Third parties must not be able to read any application data.
3. **Payload Integrity:** Changes to application data by third parties or by middleboxes with read-only access must be detectable.
4. **Visibility:** Both endpoints must be aware of all middleboxes in the session. mcTLS does not support transparent middleboxes.
5. **Least Privilege:** Each middlebox should be granted the minimum level of access needed to do its jobs.

A.2 Definitions and Notation

- mcTLS uses a pseudorandom function, constructed as in TLS 1.2, to expand secrets into blocks of key material:

  ```
  PRF(secret, label, seed)
  ```

- In this document, + indicates concatenation.
- This document introduces the idea of an encryption context. We use the terms “context” and “slice” interchangeably.

A.3 Handshake Protocol

The client generates a list of middleboxes to be used through a middlebox discovery mechanism outside the scope of this document. The client sends a TLS ClientHello message with the middlebox list included as a TLS extension. See Middlebox List Extension for the format. The ClientHello must carry the mcTLS TLS version number. When the first middlebox sees the ClientHello, it opens a TCP connection with the next middlebox and forwards the ClientHello. This continues until it reaches the server. Each middlebox examines the compression and cipher suites proposed by the client and eliminates those it does not support. The server reads the middlebox list and may immediately terminate the connection if the middlebox list violates the application requirements.

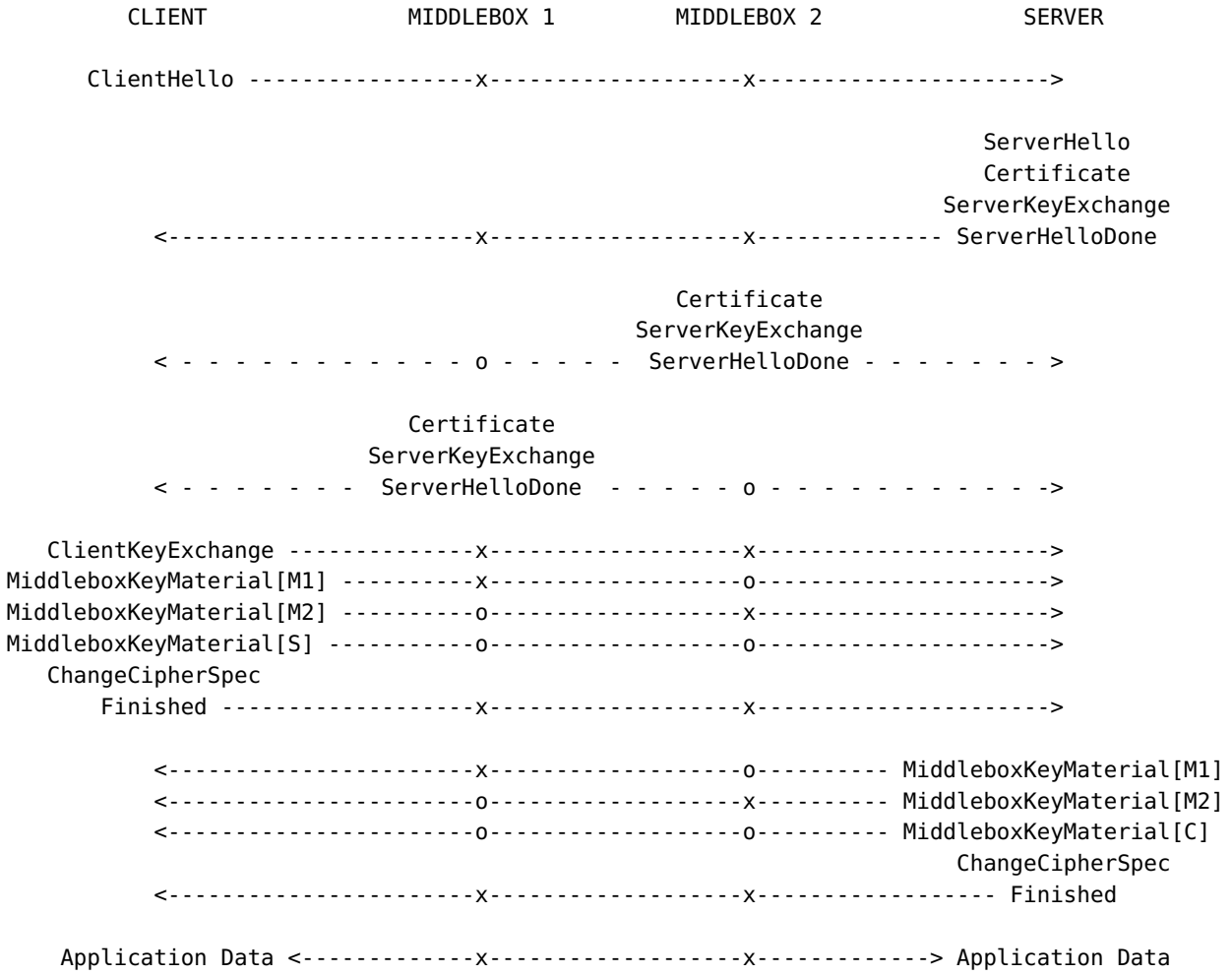
The server sends a ServerHello message specifying the compression and cipher suites to be used (i.e., the ones it picked from those offered by the client). At this point, the client will wait to receive a Certificate message from each middlebox and the server. Similarly, the server will wait until it receives a Certificate message from each middlebox. If ephemeral public key encryption is being used, then Certificate messages must be followed with ServerKeyExchange messages. Once all Certificate and ServerKeyExchange messages are received, the client sends a TLS ClientKeyExchange message to the server. With the receipt of this message, the client and server possess a shared master secret. In TLS, the master secret is used to generate the session key, which is used for encrypting and MAC-protecting application data. In mcTLS, we refer to this "session key" as the `endpoint_encryption_key` and the `endpoint_MAC_key`. The `endpoint_encryption_key` is only used for transferring context key material between the client and the server and the `endpoint_MAC_key` is used for generating each record's endpoint MAC (see Message Authentication Codes).

After distribution of the server and middlebox certificates, both the client and server generate two secrets for each context: the `client_read_secret/client_write_secret` and `server_read_secret/server_write_secret`. These secrets are used by all parties (middleboxes and endpoints) to generate the *context keys* (see Context Key Generation) that will be used by the record protocol to encrypt and MAC-protect application data.

The client and server will send MiddleboxKeyMaterial messages to each middlebox and the opposite endpoint to distribute these secrets. MiddleboxKeyMaterial is a new TLS message type. See MiddleboxKeyMaterial Message section for format. The client and server should send one another MiddleboxKeyMaterial messages containing secrets for all contexts in the middlebox_list extension. A violation must be treated as a protocol error. Middleboxes will receive secrets for each context for which they have read or write access. Once a MiddleboxKeyMaterial message arrives at its intended middlebox recipient, it must be forwarded on to the opposite end of the handshake to ensure that both the client and server observe the full sequence of handshake messages. The MiddleboxKeyMaterial message for the opposite end of the session must be sent last.

Once all MiddleboxKeyMaterial messages have been received, the client, server, and middleboxes have the encryption contexts necessary to transmit application data. First, however, validation of a successful handshake must be performed. This is accomplished by the ChangeCipherSpec message to indicate the newly negotiated encryption contexts should now be used and the transmission of a Finished message including a MAC of all handshake messages using `endpoint_MAC_key`. If the client or server fail to validate the Finished message, then the client

and server must not have observed the same sequence of handshake messages. This is a fatal error.



(In the figure, `x` indicates the middlebox reads and forwards the message; `o` indicates it just forwards it. The spaced dashed lines indicate the middlebox certificate/key exchange messages are sent piggy-backed on the `ServerKeyExchange` (toward the client) and `ClientKeyExchange` (toward the server) messages.)

A.3.1 Middlebox List Extension

mcTLS defines a new TLS handshake extension middlebox_list with tentative type ID oxff06. The extension contains:

- A list of context IDs and descriptions. A context description is a string meaningful only to the application; mcTLS does not use it.
- A list of middleboxes. Each entry specifies the middlebox's address, a unique ID, a list of the contexts for which the middlebox has read access, and a list of the contexts for which the middlebox has write access.

In order to make this list, the client application must know in advance how many contexts it will need, which middleboxes should be included, and what their permissions should be. How it

knows these things is beyond the scope of this document. The format is as follows:

```
middlebox_list {
    slice {
        slice_id
        purpose
            }[ ]
    middlebox {
        address
        middlebox_id
        read_slices[ ]
        write_slices[ ]
    }[ ]
}
```

All elements of variable size are prepended with their length. slice_id is a single byte unique identifier that will uniquely identify an encryption context (each record will be tagged with a one of these IDs—see Record Format). A slice purpose is a byte array with which the application may specify the purpose of the slice. Each middlebox is identified by an address, which is a utf8 character array containing either the domain name or IP address of the middlebox, and a middlebox_id, which is a one byte unique identifier. Each middlebox is also assigned a set of 0 or more contexts from the list slice_ids for which the middlebox has read access and a set of 0 or more slices for which the middlebox has write access. To have write access, a middlebox must also have read access. The middlebox_id values 0x00-0x02 are reserved, with 0x01 always identifying the client and 0x02 always identifying the server.

A.3.2 Middlebox Key Material Message

mcTLS introduces a new handshake message type for delivering context key material to middleboxes. During the handshake, both the client and server will send a MiddleboxKeyMaterial message to each middlebox with tentative handshake message type 0x28. The payload of the message is encrypted with a symmetric key shared between the endpoint sending the message and the middlebox receiving it and contains a "partial secret" for each context to which the middlebox has access. Once a middlebox receives and decrypts a MiddleboxKeyMaterial message from both the client and the server, it uses both secrets to generate the keys that will actually be used in the encryption of application data (see Context Key Generation). The format of the message is as follows:

```
MiddleboxKeyMaterial {
    middlebox_id
    key_material {
        slice_id
        read_secret
        write_secret
    }[ ]
}
```

A.3.3 Context Key Generation

The keys used to encrypt/decrypt and MAC-protect application data are called *context keys*. There are six symmetric keys associated with each context:

- `client_read_key`: Encrypt/decrypt data from client to server.
- `server_read_key`: Encrypt/decrypt data from server to client.
- `client_read_MAC_key`: Compute reader MAC for data from client to server.
- `server_read_MAC_key`: Compute reader MAC for data from server to client.
- `client_write_MAC_key`: Compute writer MAC for data from client to server.
- `server_write_MAC_key`: Compute writer MAC for data from server to client.

For simplicity, when referring to these keys elsewhere in this document, we do not distinguish between the client-to-server key and the server-to-client key (for example, we may simply refer to a context's `read_MAC_key`).

After receiving a MiddleboxKeyMaterial message from each endpoint, all parties compute the context keys for the contexts that they can access. For each context, each party uses the partial secrets (one from the client and one from the server) to compute two blocks of key material:

```
read_key_block = PRF(client_read_secret + server_read_secret, "reader keys",
                        client_random + server_random)
write_key_block = PRF(client_write_secret + server_write_secret, "writer keys",
                        client_random + server_random)
```

The `read_key_block` is partitioned into `client_read_key`, `server_read_key`, `client_read_MAC_key`, and `server_read_MAC_key`; the `write_key_block` is partitioned into `client_write_MAC_key` and `server_write_MAC_key`.

A.4 Record Protocol

A.4.1 Record Format

The TLS record format includes 1 byte for identifying the type of message, 2 bytes for the version of TLS in use, and 2 bytes for the message length. The header is followed by the encrypted payload including the message, MAC, and padding for block ciphers. We extend the header to include a 1 byte context ID:

```
0-------8---------------24--------------40------48---------------------------N
| TYPE  |    VERSION    |    LENGTH     | CTXT  |      PROTECTED PAYLOAD      |
-------------------------------------------------------------------------------
```

mcTLS uses a new TLS version number so that middleboxes and servers can identify an mcTLS message by decoding only the first 3 bytes of the record. Tentatively, this version number is set to 6.102.

Upon receiving an mcTLS record from the wire, the mcTLS-aware client, middlebox, or server must read the context ID value from the header and apply the correct encryption context in the decryption operation. If a middlebox does not have keys for a particular context, it should forward

the record unmodified to the next middlebox or endpoint. If a client or server does not have keys for the context, this should be treated as a protocol error.

Similarly, when generating a record for transmission on the wire, the mcTLS-aware application must specify an encryption context. The record protocol uses the corresponding keys to encrypt and MAC-protect the payload and places the context ID in the record header.

A.4.2 Message Authentication Codes

TLS uses a keyed MAC to detect message tampering by parties. mcTLS uses three MACs for each record:

- A *reader MAC*, generated with the context's `read_MAC_key`. This is used to detect changes by third parties.
- A *writer MAC*, generated with the context's `write_MAC_key`. This is used to detect (illegal) changes by middleboxes with read-only access.
- An *endpoint MAC*, generated with the `endpoint_MAC_key`. This is used to detect (legal) changes by middleboxes with write access.

The MAC format has not changed:

```
MAC_function(MAC_write_key, seq_num + record.type + record.version + record.length
                + record.content)
```

The plus sign, +, indicates concatenation. Sequence numbers are global across all encryption contexts to enforce correct ordering of all application data at the client and server. In order to maintain consistent sequence numbers across the full session, middleboxes must never modify the number or order of mcTLS records on the communication medium.

The order of the MACs after the record payload is not significant and chosen arbitrarily to be: payload, reader MAC, writer MAC, endpoint MAC.

A.5 Application Programming Interface

To enable mcTLS applications, the TLS API must be expanded to include new methods for specifying the encryption context in use, handling of the MAC, and specifying the middleboxes to use in a handshake. The following methods have been added to the API:

```
int          mcTLS_connect(SSL *ssl, mcTLS_SLICE* slices, int slices_len, mcTLS_PROXY *middleboxes, int middleboxes_len);
int          mcTLS_middlebox(SSL *ssl, SSL* ( *connect_func)(SSL *ssl, char *address), SSL **ssl_next);
int          mcTLS_get_slices(SSL *ssl, mcTLS_SLICE **slices, int *slices_len);
int          mcTLS_get_middleboxes(SSL *ssl, mcTLS_PROXY **middleboxes, int *middleboxes_len);
mcTLS_PROXY* mcTLS_generate_middlebox(SSL *s, char* address);
mcTLS_PROXY* mcTLS_middlebox_from_id(SSL *ssl, int middlebox_id); **TO BE REMOVED**
mcTLS_SLICE* mcTLS_generate_slice(SSL *s, char* purpose);
mcTLS_SLICE* mcTLS_slice_from_id(SSL *ssl, int slice_id);  **TO BE REMOVED**
int          mcTLS_assign_middlebox_write_slices(SSL *s, mcTLS_PROXY* middlebox, mcTLS_SLICE* slices[ ], int slices_len);
int          mcTLS_assign_middlebox_read_slices(SSL *s, mcTLS_PROXY* middlebox, mcTLS_SLICE* slices[ ], int slices_len);
int          mcTLS_read_record(SSL *ssl, void *buf, int num, mcTLS_SLICE **slice, mcTLS_CTX **mac);
int          mcTLS_write_record(SSL *ssl, const void *buf, int num, mcTLS_SLICE *slice);
int          mcTLS_forward_record(SSL *ssl, const void *buf, int num, mcTLS_SLICE *slice, mcTLS_CTX *mac, int modified);
```

`mcTLS_connect(...)` expands on the standard TLS connect method to allow clients to specify the middlebox list to be included in the extension of the handshake.

`mcTLS_generate_middlebox(...)` is used by the client to generate the state for a middlebox. This should be called once for each middlebox in the session and the resulting state objects passed to `mcTLS_connect(...)`.

`mcTLS_generate_slice(...)` is used by the client to initialize the state for each slice to be used in a session. It should be called once be slice and the resulting state objects passed as an array to `mcTLS_connect(...)`.

`mcTLS_assign_middlebox_write_slices(...)` and `mcTLS_assign_middlebox_read_slices(...)` are used to specify the slices that a middlebox has read and write access to. Setting these will control the distribution of encryption contexts during the handshake. Write access for a middlebox implies read access.

`mcTLS_middlebox(...)` is the equivalent call of `mcTLS_connect(...)` for a middlebox implementation. Upon receiving a connection from a client, the middlebox will instantiate an instance of the SSL library and call `mcTLS_middlebox(...)` passing a callback function. The callback function will be called during the handshake to enable the application to make a connection to the next middlebox or the server and create another SSL library instance for the second connection.

`mcTLS_read_record(...)` is used by clients, servers, and middleboxes to read the next available record from the communication medium. If there is no record available, the return value is 0. slice returns the encryption context used to decrypt the message or an empty encryption context if the specified encryption context is not available to the middlebox. If an encryption context is not available at either the client or the server, it should be treated as a fatal error.

`mcTLS_write_record(...)` is used by clients and servers. The message is encrypted using the specified encryption context and written to the communication medium. If the encryption context is not available, an error is returned.

`mcTLS_forward_record(...)` is used by mcTLS middleboxes to send a record on after processing. If the encryption context is not available, then the application data passed is treated as an encrypted record and forwarded without further processing. If modified is false, the specified MAC is used and the message is then encrypted. If modified is true, the specified MAC value is ignore and a new MAC will be generated before encryption. Because the MAC generation includes a sequence number, the middlebox must insure that the number of sent and received records remains consistent and ordered upon each side of the middlebox.

`mcTLS_SLICE` is a new structure that includes the full encryption context of a traditional TLS session. The mcTLS client, server, and middlebox must maintain one structure for each encryption context negotiated for the session. The `mcTLS_SLICE` structure includes a slice_id 1 byte value used to identify the slice within the mcTLS record format and a have_material boolean value that indicates whether this client, server, or middlebox has the encryption context. Clients and servers should treat the receipt of any message for a slice where have_material is not true as a protocol error. The structure has the following members which are relevant to the application.

```
mcTLS_SLICE {
    int read_access;
    int write_access;
    char *purpose;
}
```

`read_access` and `write_access` are boolean values indicating whether the middlebox has

read or write access to the slice, respectively. Clients and servers always have both read and write access to all slices. Purpose is a null terminated string that is defined by the client before the handshake to assign an application dependent meaning to the slice. It can be used to indicate what type of application data should be pass with each slice.

`mcTLS_CTX` is context information from a previous call to `mcTLS_read_record(...)`. Middlebox implementations should pass the context to the related `mcTLS_forward_record(...)` call. Client and server implementations may ignore this value.

Appendix B mcTLS Security Analysis

B.1 Handshake Protocol

The mcTLS handshake is ultimately responsible for two things:

1. **Distributing context keys to the correct entities.** (The context keys are used to encrypt/decrypt/MAC application data, so if only the correct entities have the context keys then only the correct entities can access/alter application data.)
2. **Negotiating session configuration parameters between the client and ther server.** *[cipher suite, session path (list of middleboxes + server), number of contexts, middleboxes permissions for each context].* (Doing this successfully implies that no one but the client and server—including middleboxes—can force the session to use a weak encryption cipher or give middleboxes higher permissions than the application intended.)

1. Distributing Context Keys Like TLS, mcTLS can operate in multiple authentication modes. Any party may optionally authenticate any other party, with the exception that middleboxes never authenticate one another. As we argue below, as long as each middlebox and the server are authenticated by at least one endpoint (though clearly the client must be the one to authenticate the server), a man-in-the-middle attack is not possible. In particular, *it is sufficient for the client to authenticate each middlebox and the server*; the middleboxes and the server do not need to authenticate anyone. For simplicity, in the rest of this document we assume that the client authenticates everyone.

Since the client sends partial context secrets to each party encrypted under a symmetric key it shares with that party, *it is sufficient to show that each symmetric key the client establishes is really shared by the correct entity (see below).* (Even if the server does not authenticate anyone and sends partial context secrets to an adversary, the server's partial secrets are useless without secrets from the client as well.)

2. Session Configuration Many configuration parameters are exchanged before any parties have established shared keys, so they are sent in the clear. These values' being visible is not a security risk, but an attacker could weaken the security of the session by actively modifying these messages in flight (e.g., to make the endpoints pick a weaker encryption algorithm than they ordinarily would).

Endpoints: If handshake messages are modified, the client and server will compute different handshake transcript hashes and will abort the session when they notice the discrepancy. Since these hashes are protected by `endpoint_MAC_key`, an adversary cannot update them to reflect any changes it made to earlier handshake messages. Therefore, to show that configuration parameters

are safe, *it is sufficient to show that the client establishes a shared key with the correct server (see below).*

Middleboxes: mcTLS does not give middleboxes a way to verify the handshake transcript. This means an adversary could arbitrarily alter any handshake message sent to a middlebox. Middleboxes forward all handshake messages to the opposite endpoint, but the adversary could drop the forwarded modified message and replay the correct message to the endpoint, making this attack undetectable by the endpoints when they exchange Finished messages. For each piece of information carried by the handshake in the clear, we argue that the security of the session is not compromised if the middlebox cannot verify it.

- *Cipher Suite:* The endpoints pick a cipher suite without input from middleboxes. If the cipher suite is modified by anyone (attacker or middlebox), the endpoints will detect the change. Attackers cannot influence the choice of cipher suite by modifying the cipher suite right before the ClientHello/ServerHello passes through a middlebox and correcting the change before it is forwarded to an endpoint. In this case, the middlebox will not be able to decrypt application data, but this is a denial of service attack (no different from the adversary dropping all packets to that middlebox).
- *Number of Contexts:* Same argument.
- *Middlebox Context Permissions:* Same argument.
- *Session Path:* The session path is more subtle because middleboxes actually use this information during the handshake process—it tells each one what "next hop" to connect to. An adversary could replace the last hop in the path—the intended server S—with an alternate server, S'. The last middlebox in the path, M, would connect to S' and successfully establish a key with it. (This works even if M authenticates the server, because authentication depends on checking a certificate, which requires that you know the domain name of the correct party. In this case, the adversary has altered the domain name in the session path, making M think that S' is the correct server.) Meanwhile, the adversary sends S copies of all of the expected handshake messages, so the client and the server do not detect that anything is amiss. However, this attack does not succeed: S' does not learn the context keys. This is because middleboxes never use the shared keys they establish with each endpoint to forward context key material; they only use them to receive context key material. So, S' can send bogus context keys to M (a DoS attack). And even though the client sends legitimate context keys to M, M never forwards these to S' encrypted under a key S' knows.

Key Exchange So far we have determined that the mcTLS handshake succeeds in its two goals if the client successfully establishes a shared key with each party (known only to the client and that party). Here we argue that this is the case:

Client-Server Key Exchange: The client and server use a standard TLS KE mechanism to derive a shared secret. Although mcTLS adds extra information to these handshake messages for other purposes, the information related to client-server KE is unchanged and is used exactly as it is used in TLS. Therefore, with respect to client-server KE, mcTLS inherits TLS' security properties. (The fact that these messages are now forwarded via one or more middleboxes is irrelevant; as far as client-server KE is concerned, the middleboxes are just like any other third-party entity that

TLS was designed to protect against.)

Client-Middlebox Key Exchange: mcTLS adds client-middlebox key exchanges to the TLS handshake. These KEs are performed using standard TLS KE mechanisms. By considering the middleboxes one at a time, the messages exchanged between the client and each middlebox look like a standard client-server TLS KE (and are generated/used just as they would be if the middlebox were a server in a normal TLS KE). Therefore, the client establishes a shared secret with each middlebox with the same security properties as a normal TLS KE.

Note: It may seem strange that the middleboxes do not need to authenticate anyone. This is because mcTLS is designed with the philosophy that the session "belongs to" the endpoints. The endpoints decide which middleboxes should be able to access/alter application data and mcTLS enforces this. In contrast, middlebox A cannot ensure that middlebox B does or does not have access to a session; we view this as a policy issue, not a security property for mcTLS to guarantee.

B.2 Record Protocol

The mcTLS record protocol carries data which has been assigned by the application to one of multiple encryption contexts. Each context has two keys: a read key and a write key. The endpoints also share a key (the endpoint key) that is not particular to any one context. Before sending a record, mcTLS MAC-protects and encrypts it in the same way TLS would, except that mcTLS encrypts using the context's read key and generates three MACs, one for each key. The handshake protocol ensures that the correct parties have the correct context keys.

Read/Write Access Which contexts an entity can read or write depends on which keys it has.

- *Third parties.* The handshake ensures that third parties learn no context keys, so they cannot decrypt application data or recompute MACs (so they can neither read nor write). The same security properties that apply to third parties in TLS apply in mcTLS.
- *Readers.* Middleboxes with read-only permission for a context are given only the read key. This means they can decrypt the data, but cannot recompute the writer or endpoint MACs, so they do not have write access. (More on checking MACs below.)
- *Writers.* Middleboxes with read+write permission for a context have both the read key and the write key, meaning they can decrypt (read access) and recompute the write MAC (write access).
- *Endpoints.* Endpoints know all context keys, so they have full read+write access to the data stream.

Detecting Illegal Writes: "Illegal reads" simply cannot happen; parties without read permission do not know the decryption key. Illegal writes (i.e., modifications by middleboxes with read-only access) are possible in the sense they can use the read key to decrypt, change, and re-encrypt application data. mcTLS' endpoint-writer-reader MAC scheme allows endpoints and writers to detect such modifications.

Readers

- **Can** detect **third party** changes because each record includes a MAC computed with the read key. (Endpoints and writers generate this MAC whenever they modify a record.)

- **Cannot** detect changes by **other readers** because all readers know the read key, so another reader could make an illegal change and generate a new, valid read MAC. Developers of middlebox software should be aware of this limitation and take appropriate precautions if illegal changes by another reader could pose a security risk to their application.
- **Cannot** detect **writer** changes because they do not know the write key (and so cannot verify the writer MAC). This is okay, because a change by a writer is not a security violation.

Writers

- **Can** detect **third party** changes by checking the reader MAC.
- **Can** detect **reader** changes by checking the writer MAC.
- **Cannot** detect changes by **other writers** because all writers know the write key, so they can generate a new, valid writer MAC. Again, a change by a writer is not a security violation.

Endpoints

- **Can** detect **third party** changes by checking the reader MAC.
- **Can** detect **reader** changes by checking the writer MAC.
- **Can** detect **writer** changes by checking the endpoint MAC. Again, a writer change is not a security violation, but the endpoint application may be curious whether or not the record was changed by a middlebox.

Dropping, Reordering, or Replaying Records Like TLS, records in mcTLS carry sequence numbers (included in the MACs) to prevent dropping, reordering, or replaying records. In mcTLS, sequence numbers are global across contexts, otherwise attackers could delete a context entirely or drop the last record in a context without detection. Like TLS, separate encryption/MAC keys in each direction prevent replaying a record from one direction in the other direction.

Appendix C mbTLS Protocol Details

This appendix describes the message formats and protocol constants used in our implementation of mbTLS. It follows the formatting conventions set forth in the RFC 5246 (TLS 1.2) [94].

C.1 Record Protocol

mbTLS adds three new record types (bold):

```
enum {
  change_cipher_spec(20),
  alert(21),
  handshake(22),
  application_data(23),
  mbtls_encapsulated(30),
  mbtls_key_material(31),
  mbtls_middlebox_announcement(32),
  (255)
} ContentType;
```

Encapsulated. The `MBTLSEncapsulated` record is used to carry secondary handshake messages.

```
struct {
    uint8 subchannelId;
    opaque record[TLSPlaintext.length-1];
} EncapsulatedRecord;
```

Here, `record` is another complete TLS record. Because the inner record is the outer recordâĂŹs payload, which is limited to 2^14 bytes, and because the subchannel ID uses 1 byte, the inner record's payload is limited to 2^{14-1} bytes. `MBTLSEncapsulated` records may only be sent during handshake or renegotiation.

Key Material. The `MBTLSKeyMaterial` record is used by endpoints to send symmetric key material to their middleboxes.

```
struct {
    uint32 key_len;
    uint32 iv_len;
    opaque clientWriteKey[key_len];
    opaque clientWriteIV[iv_len];
    opaque clientReadKey[key_len];
    opaque clientReadIV[iv_len];
    opaque serverWriteKey[key_len];
    opaque serverWriteIV[iv_len];
    opaque serverReadKey[key_len];
    opaque serverReadIV[iv_len];
} VMPGCMKeyMaterial;

struct {
    Version client_server_version;
    opaque client_to_server_sequence[8];
    opaque server_to_client_sequence[8];
    CipherSuite cipher_suite;  /* 2 bytes */
    select(cipher_suite) {
        case TLS_RSA_WITH_AES_256_GCM_SHA384:
          VMPGCMKeyMaterial;
    }
} VMPKeyMaterial;
```

The `MBTLSKeyMaterial` message is always sent encapsulated in a subchannel (i.e., in an `MBTLSEncapsulated` record). It contains the TLS version negotiated between the client and the server, the sequence number for client-to-server data (write sequence from client's perspective and read sequence from server's) and the sequence number for server-to-client data. It also contains key and IV material in a format dependent on the cipher suite.

Middlebox Announcement. The `MBTLSMiddleboxAnnouncement` message is used by middleboxes to alert the server to their presence.

```
struct {
} VMPMiddleboxAnnouncement;
```

The `MBTLSMiddleboxAnnouncement` message is always sent encapsulated in a subchannel (i.e., in an `MBTLSEncapsulated` record). The message is empty, and only serves to alert the server of the middlebox's presence. Middleboxes never send this message to a client; to announce themselves to a client, they simply respond to the `ClientHello` in the `MiddleboxSupportExtension` described below.

C.2 Handshake Protocol

mbTLS adds one handshake protocol message (bold):

```
enum {
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    sgx_attestation(17),
    finished(20), (255)
} HandshakeType;
```

SGX Attestation. The `SGXAttestation` handshake message can be optionally used during the handshake for the server to send the client an SGX attestation (quote). This feature is independent of the rest of mbTLS.

```
struct {
    opaque sgx_quote<0..2^14-1>;
} SGXAttestation;
```

`sgx_quote` follows Intel's `sgx_quote_t` format.

Middlebox Support Extension. mbTLS also adds one TLS extension, the `MiddleboxSupportExtension`:

```
struct {
    uint8 numHellos;
    uint16 helloLengths[numHellos];
    opaque clientHellos[numHellos];
} MiddleboxSupportExtension;
```

The `MiddleboxSupportExtension` is sent by a TLS client in the `ClientHello` message. It indicates that the client supports mbTLS, inviting on-path middleboxes to announce themselves to the client. The extension carries one or more "optimistic" `ClientHello`s, to which the middleboxes may respond with `ServerHello`s.

Bibliography

[1] https://mctls.org.

[2] http://www.rubymotion.com/.

[3] https://github.com/scoky/mctls.

[4] Anonymizer. https://www.anonymizer.com.

[5] ARM TrustZone. https://www.arm.com/products/security-on-arm/trustzone. Accessed: January 2017.

[6] Aryaka. http://www.aryaka.com. Accessed: January 2017.

[7] Brocade Network Functions Virtualization. http://www.brocade.com/en/products-services/software-networking/network-functions-virtualization.html. Accessed: January 2017.

[8] curl. https://curl.haxx.se.

[9] Dashboards—android developers. https://developer.android.com/about/dashboards/index.html. Accessed: Dec. 2014.

[10] Data compression proxy. https://developer.chrome.com/multidevice/data-compression. Accessed: Dec. 2014.

[11] Disconnect. https://disconnect.me.

[12] Expressvpn. https://www.expressvpn.com.

[13] Ghostery. https://www.ghostery.com.

[14] Google Edge Network. https://peering.google.com. Accessed: January 2017.

[15] Hide my ass! https://www.hidemyass.com.

[16] Intel Official SGX OpenSSL Library. https://software.intel.com/en-us/sgx-sdk/download. Accessed: June 2017.

[17] Juniper Architecture for Technology Transformation. https://www.juniper.net/assets/us/en/local/pdf/whitepapers/2000633-en.pdf. Accessed: January 2017.

[18] mbTLS. http://mbtls.org.

[19] Mining Hardware Comparison. https://en.bitcoin.it/wiki/Mining_hardware_comparison.

[20] Network Functions Virtualization (Dell). http://www.dell.com/en-us/work/learn/tme-telecommunications-solutions-telecom-nfv. Accessed: January 2017.

[21] PhantomJS. http://phantomjs.org.

[22] Strongvpn. https://strongvpn.com.

[23] Surveillance Self-Defense. https://ssd.eff.org.

[24] Telefónica NFV Reference Lab. http://www.tid.es/long-term-innovation/network-innovation/telefonica-nfv-reference-lab. Accessed: January 2017.

[25] TLS for SGX: a port of mbedtls. https://github.com/bl4ck5un/mbedtls-SGX. Accessed: June 2017.

[26] Tor Project—Anonymity Online. https://www.torproject.org.

[27] University of Oregon Route Views Project. http://www.routeviews.org.

[28] Users – tor metrics. https://metrics.torproject.org/userstats-relay-country.html.

[29] Vyprvpn. https://www.goldenfrog.com/vyprvpn.

[30] Web profiler. https://github.com/dtnaylor/web-profiler.

[31] Zscaler. https://www.zscaler.com. Accessed: January 2017.

[32] Wikipedia qatar ban 'temporary'. http://news.bbc.co.uk/2/hi/technology/6224677.stm, January 2007.

[33] AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about_us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf, 2013. Accessed: January 2017.

[34] ETSI TR 103 421. http://www.etsi.org/deliver/etsi_tr/103400_103499/103421/01.01.01_60/tr_103421v010101p.pdf, April 2017.

[35] Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, 2017. USENIX Association.

[36] André Adelsbach, Ulrich Greveler, Stephan Groß, and Sandra Steinbrecher. Anocast: Rethinking broadcast anonymity in the case of wireless communication. In *Sicherheit*, volume 128, pages 71–84, 2008.

[37] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. Flywheel: Google's data compression proxy for the mobile web. NSDI '15, pages 367–380, Oakland, CA, May 2015. USENIX Association.

[38] Georgios Amanatidis, Alexandra Boldyreva, and Adam O'Neill. Provably-secure schemes for basic query support in outsourced databases. In *Proceedings of the 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 14–30, Berlin, Heidelberg, 2007. Springer-Verlag.

[39] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP '13*, volume 13, 2013.

[40] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon,

and Scott Shenker. Accountable internet protocol (AIP). SIGCOMM '08, pages 339–350, New York, NY, USA, 2008. ACM.

[41] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, GA, 2016. USENIX Association.

[42] G. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How Much Does It Really Cost? In *INFOCOM 1999*, 1999.

[43] Katerina J Argyraki and David R Cheriton. Active internet traffic filtering: Real-time response to denial-of-service attacks. In *USENIX Annual Technical Conference, General Track*, pages 135–148, 2005.

[44] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI '16*, pages 689–703, GA, 2016. USENIX Association.

[45] Ash de Zylva. Windows 10 Device Guard and Credential Guard Demystified. https://blogs.technet.microsoft.com/ash/2016/03/02/windows-10-device-guard-and-credential-guard-demystified/. Accessed: January 2017.

[46] Pierre-Louis Aublin, Florian Kelbert, Dan OâĂŹKeeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves. 2017.

[47] T. Aura. Cryptographically Generated Addresses (CGA). RFC 3972 (Proposed Standard), March 2005. Updated by RFCs 4581, 4982.

[48] F. Baker and P. Savola. Ingress Filtering for Multihomed Networks. RFC 3704 (Best Current Practice), March 2004.

[49] Feng Bao, Robert H. Deng, Xuhua Ding, and Yanjiang Yang. Private query on encrypted data in multi-user settings. In *Proceedings of the 4th International Conference on Information Security Practice and Experience*, ISPEC'08, pages 71–85, Berlin, Heidelberg, 2008. Springer-Verlag.

[50] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI '14*. USENIX – Advanced Computing Systems Association, October 2014.

[51] Doug Beaver. HTTP2 Expression of Interest. http://lists.w3.org/Archives/Public/ietf-http-wg/2012JulSep/0251.html, 7 2012.

[52] Adam Bender, Neil Spring, Dave Levin, and Bobby Bhattacharjee. Accountability as a service. *SRUTI*, 7:1–6, 2007.

[53] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[54] Oliver Berthold, Hannes Federrath, and Stefan Köpsell. Web mixes: A system for anonymous and unobservable internet access. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, pages 115–129, New York, NY,

USA, 2001. Springer-Verlag New York, Inc.

[55] Oliver Berthold, Andreas Pfitzmann, and Ronny Standtke. The disadvantages of free mix routes and how to overcome them. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 30–45. Springer Berlin Heidelberg, 2001.

[56] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, May 2015.

[57] Mohit Bhargava and Catuscia Palamidessi. Probabilistic anonymity. In MartÃČÂŋn Abadi and Luca de Alfaro, editors, *CONCUR 2005—Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin Heidelberg, 2005.

[58] K. Bhargavan, C. Fournet, and M. Kohlweiss. mitls: Verifying protocol implementations against real-world attacks. *IEEE Security Privacy*, 14(6):18–25, Nov 2016.

[59] Karthikeyan Bhargavan, IC Boureanu, Pierre-Alain Fouque, Cristina Onete, and Benjamin Richard. Content delivery over tls: A cryptographic analysis of keyless ssl. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, 2017.

[60] Karthikeyan Bhargavan, Cedric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing tls with verified cryptographic security. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 445–459, Washington, DC, USA, 2013. IEEE Computer Society.

[61] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. *Proving the TLS Handshake Secure (As It Is)*, pages 235–255. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[62] A. Bittau, D. Boneh, M. Hamburg, M. Handley, D. Mazieres, and Q. Slack. Cryptographic protection of tcp streams (tcpcrypt), February 2014.

[63] 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.

[64] Matt Blaze, John Ioannidis, Angelos D Keromytis, Tal G Malkin, and Avi Rubin. Anonymity in wireless broadcast networks. *International Journal of Network Security*, 8(1):37–51, 2009.

[65] Marcelo Bagnulo Braun and Jon Crowcroft. SNA: Sourceless Network Architecture. Technical Report UCAM-CL-TR-849, University of Cambridge, Computer Laboratory, March 2014.

[66] Jonah Engel Bromwich. Protecting Your Digital Life in 9 Easy Steps. https://www.nytimes.com/2016/11/17/technology/personaltech/encryption-privacy.html, November 2016.

[67] Ian Brown. End-to-end security in active networks. In *University College London PhD Thesis*, 2001.

[68] Philip Bump. So, You Want to Hide from the NSA? Your Guide to the Nearly Impossible. https://www.theatlantic.com/technology/archive/2013/07/so-you-want-

hide-nsa-your-guide-nearly-impossible/313510/, July 2013.

[69] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Advances in Cryptology-EUROCRYPT 2001*, pages 93–118. Springer, 2001.

[70] David Cameron. The internet and pornography: Prime minister calls for action. https://www.gov.uk/government/speeches/the-internet-and-pornography-prime- minister-calls-for-action. Accessed: Jan. 2015.

[71] Catharina Candolin and Pekka Nikander. IPv6 source addresses considered harmful. In *NordSec '01*, pages 54–68, 2001.

[72] C. Castelluccia. Improving Secure Server Performance by Rebalancing SSL/TLS Handshakes. In *USENIX Security Symposium*, 2005.

[73] Rohit Chadha, StÃČÂľphanie Delaune, and Steve Kremer. Epistemic logic for the applied pi calculus. In David Lee, AntÃČÂşnia Lopes, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems*, volume 5522 of *Lecture Notes in Computer Science*, pages 182–197. Springer Berlin Heidelberg, 2009.

[74] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. In *Trustworthy Global Computing*, pages 281–300. Springer, 2007.

[75] D Chaum. Untraceable electronic mail, return address, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.

[76] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, March 1988.

[77] Chen Chen, Daniele E. Asoni, David Barrera, George Danezis, and Adrain Perrig. Hornet: High-speed onion routing at the network layer. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1441–1454, New York, NY, USA, 2015. ACM.

[78] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. RFC 6763 (Proposed Standard), February 2013.

[79] S. Cheshire and M. Krochmal. Multicast DNS. RFC 6762 (Proposed Standard), February 2013.

[80] David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden. Tussle in cyberspace: defining tomorrow's internet. SIGCOMM '02, pages 347–356, New York, NY, USA, 2002. ACM.

[81] C. Coarfa, P. Druschel, and D. S. Wallach. Performance Analysis of TLS Web Servers. *ACM Trans. Comput. Syst.*, 24(1):39–69, February 2006.

[82] David Cole. 'We Kill People Based on Metadata'. http://www.nybooks.com/daily/2014/05/10/we-kill-people-based-metadata/, May 2014.

[83] Adrian Colyer. Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. https://blog.acolyer.org/2016/06/20/multi-context-tls-mctls-

enabling-secure-in-network-functionality-in-tls/, June 2016.

[84] Michael Cooney. IBM touts encryption innovation: New technology performs calculations on encrypted data without decrypting it. http://www.computerworld.com/article/2526031/security0/ibm-touts-encryption-innovation.html, June 2009.

[85] Henry Corrigan-Gibbs, Dan Boneh, and David MaziÃĺres. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 321–338, Washington, DC, USA, 2015. IEEE Computer Society.

[86] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 340–350, New York, NY, USA, 2010. ACM.

[87] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in verdict. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 147–162, Washington, D.C., 2013. USENIX.

[88] George Danezis, Claudia Diaz, and Paul Syverson. Systems for anonymous communication, 2009.

[89] George Danezis, Claudia Diaz, Carmela Troncoso, and Ben Laurie. Drac: An architecture for anonymous low-volume communications. In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, PETS'10, pages 202–219, Berlin, Heidelberg, 2010. Springer-Verlag.

[90] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 2–, Washington, DC, USA, 2003. IEEE Computer Society.

[91] Yuxin Deng, Jun Pang, and Peng Wu. Measuring anonymity with relative entropy. In *Formal Aspects in Security and Trust*, pages 65–79. Springer, 2007.

[92] Claudia Díaz, Stefaan Seys, Joris Claessens, and Bart Preneel. Towards measuring anonymity. In Roger Dingledine and Paul Syverson, editors, *Privacy Enhancing Technologies*, volume 2482 of *Lecture Notes in Computer Science*, pages 54–68. Springer Berlin Heidelberg, 2003.

[93] Claudia Diaz, Carmela Troncoso, and Andrei Serjantov. On the impact of social network profiling on anonymity. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies*, PETS '08, pages 44–62, Berlin, Heidelberg, 2008. Springer-Verlag.

[94] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[95] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov 1976.

[96] R Dingeldine, Nick Mathewson, and P Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[97] Fahad R. Dogar, Peter Steenkiste, and Konstantina Papagiannaki. Catnap: exploiting high bandwidth wireless interfaces to save energy for mobile devices. MobiSys '10, pages 107–122, New York, NY, USA, 2010. ACM.

[98] Zhenhai Duan, Xin Yuan, and Jaideep Chandrashekar. Constructing inter-domain packet filters to control ip spoofing based on bgp updates. In *INFOCOM*, 2006.

[99] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. The Security Impact of HTTPS Interception. In *NDSS '17*, 2017.

[100] Kit Eaton. How One Second Could Cost Amazon $1.6 Billion In Sales. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales, March 2012.

[101] Matthew Edman, Fikret Sivrikaya, and Bülent Yener. A combinatorial approach to measuring anonymity. In *Intelligence and Security Informatics, 2007 IEEE*, pages 356–363. IEEE, 2007.

[102] Yehia El-khatib, Gareth Tyson, and Michael Welzl. Can SPDY Really Make the Web Faster? In *IFIP Networking 2014*, 2014.

[103] J. Erman, A. Gerber, M. Hajiaghayi, Dan Pei, S. Sen, and O. Spatscheck. To cache or not to cache: The 3g case. *Internet Computing, IEEE*, 15(2):27–34, March 2011.

[104] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan. Towards a SPDY'Ier Mobile Web? In *CoNEXT '13*, 2013.

[105] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. The Locator/ID Separation Protocol (LISP). RFC 6830 (Experimental), January 2013.

[106] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *NSDI 14*, pages 543–546, Seattle, WA, 2014. USENIX Association.

[107] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827 (Best Current Practice), May 2000. Updated by RFC 3704.

[108] David Fifield and Serge Egelman. *Fingerprinting Web Users Through Font Metrics*, pages 107–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[109] A. Finamore, M. Mellia, M. Meo, M. M. Munafò, and D. Rossi. Experiences of Internet Traffic Monitoring with Tstat. *IEEE Network*, 25(3), 2011.

[110] Jun Furukawa and Kazue Sako. An efficient scheme for proving a shuffle. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 368–387, London, UK, UK, 2001. Springer-Verlag.

[111] Flavio D. Garcia, Ichiro Hasuo, Wolter Pieters, and Peter van Rossum. Provable anonymity. In *Proceedings of the 2005 ACM Workshop on Formal Methods in Security Engineering*, FMSE '05, pages 63–72, New York, NY, USA, 2005. ACM.

[112] Joseph Gardiner and Shishir Nagaraja. Blindspot: Indistinguishable anonymous communications. *CoRR*, abs/1408.0784, 2014.

[113] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science*,

FOCS '13, pages 40–49, Washington, DC, USA, 2013. IEEE Computer Society.

[114] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[115] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[116] Benedikt Gierlichs, Carmela Troncoso, Claudia Diaz, Bart Preneel, and Ingrid Verbauwhede. Revisiting a combinatorial approach toward measuring anonymity. In *Proceedings of the 7th ACM Workshop on Privacy in the Electronic Society*, WPES '08, pages 111–116, New York, NY, USA, 2008. ACM.

[117] Sharad Goel, Mark Robson, Milo Polte, and Emin Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical report, Cornell University, 2003.

[118] David M Goldschlag, Michael G Reed, and Paul F Syverson. Hiding routing information. In *Information Hiding*, pages 137–150. Springer, 1996.

[119] Philippe Golle and Ari Juels. Dining Cryptographers Revisited. In Christian Cachin and Jan L Camenisch, editors, *EUROCRYPT 2004*, pages 456–473. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[120] Andy Greenberg. DARPA Will Spend $20 Million To Search For Crypto's Holy Grail. https://www.forbes.com/sites/andygreenberg/2011/04/06/darpa-will-spend-20-million-to-search-for-cryptos-holy-grail, April 2011.

[121] Saikat Guha, Bin Cheng, and Paul Francis. Privad: Practical privacy in online advertising. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 169–182, Berkeley, CA, USA, 2011. USENIX Association.

[122] Ceki Gulcu and Gene Tsudik. Mixing email with babel. In *Proceedings of the 1996 Symposium on Network and Distributed System Security (SNDSS '96)*, SNDSS '96, pages 2–, Washington, DC, USA, 1996. IEEE Computer Society.

[123] P. Guy. Not as SPDY as You Thought. http://goo.gl/RQkTwx, June 2012.

[124] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2. Internet-Draft draft-tsvwg-quic-protocol-02, IETF Secretariat, January 2016.

[125] Dongsu Han, Ashok Anand, Fahad Dogar, Boyan Li, Hyeontaek Lim, Michel Machado, Arvind Mukundan, Wenfei Wu, Aditya Akella, David G. Andersen, John W. Byers, Srinivasan Seshan, and Peter Steenkiste. XIA: efficient support for evolvable internetworking. NSDI'12, pages 23–23, Berkeley, CA, USA, 2012. USENIX Association.

[126] Seungyeop Han, Vincent Liu, Qifan Pu, Simon Peter, Thomas Anderson, Arvind Krishnamurthy, and David Wetherall. Expressive privacy control with pseudonyms. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 291–302, New York, NY, USA, 2013. ACM.

[127] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard),

November 1998. Obsoleted by RFC 4306, updated by RFC 4109.

[128] Alex Hern. Worried about the NSA under Trump? Here's how to protect yourself. https://www.theguardian.com/technology/2016/nov/10/nsa-trump-protect-yourself, November 2016.

[129] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP '13*, page 11, 2013.

[130] M.A. Hoque, M. Siekkinen, and J. K. Nurminen. On the energy efficiency of proxy-based traffic shaping for mobile audio streaming. In *Consumer Communications and Networking Conference (CCNC)*, pages 891–895. IEEE, 2011.

[131] Hsu-Chun Hsiao, TH-J Kim, Adrian Perrig, Akira Yamada, Samuel C Nelson, Marco Gruteser, and Wei Meng. Lap: Lightweight anonymity and privacy. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 506–520. IEEE, 2012.

[132] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged ssl certificates in the wild. In *IEEE Symposium on Security and Privacy*, SP '14, pages 83–97, Washington, DC, USA, 2014. IEEE Computer Society.

[133] Dominic Hughes and Vitaly Shmatikov. Information hiding, anonymity and privacy: a modular approach. *Journal of Computer security*, 12(1):3–36, 2004.

[134] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 533–549, GA, 2016. USENIX Association.

[135] IETF HTTPbis Working Group. Http/2. http://http2.github.io/.

[136] Sunghwan Ihm and Vivek S. Pai. Towards Understanding Modern Web Traffic. In *IMC 2011*, 2011.

[137] Subodh Iyengar and Kyle Nekritz. Building Zero protocol for fast, secure mobile connections. https://code.facebook.com/posts/608854979307125/building-zero-protocol-for-fast-secure-mobile-connections/, January 2017.

[138] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM.

[139] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM.

[140] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *NSDI 2011*, 2011.

[141] Cheng Jin, Haining Wang, and Kang G Shin. Hop-count filtering: an effective defense against spoofed ddos traffic. In *CCS '03*, pages 30–41. ACM, 2003.

[142] V.P. Kafle, K. Nakauchi, and M. Inoue. Generic identifiers for id/locator split internetworking.

In *K-INGN 2008.*, pages 299–306, 2008.

[143] Srikanth Kandula, Dina Katabi, Matthias Jacob, and Arthur Berger. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *NSDI '05*, pages 287–300. USENIX Association, 2005.

[144] Sneha Kasera, Semyon Mizikovsky, Ganapathy S. Sundaram, and Thomas Y. C. Woo. On securely enabling intermediary-based services and performance enhancements for wireless mobile users. In *in Workshop on Wireless Security, 2003*, pages 61–68, 2003.

[145] Sachin Katti, Dina Katabi, and Katarzyna Puchala. Slicing the Onion: Anonymous Routing Without PKI. Technical Report MIT-CSAIL-TR-2005-053, MIT, August 2005.

[146] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. NSDI '12, pages 113–126, San Jose, CA, 2012. USENIX.

[147] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005. Updated by RFCs 6040, 7619.

[148] T. Killalea. Recommended Internet Service Provider Security Services and Procedures. RFC 3013 (Best Current Practice), November 2000.

[149] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A first step towards leveraging commodity trusted execution environments for network applications. In *HotNets-XIV*, pages 7:1–7:7, New York, NY, USA, 2015. ACM.

[150] Lea Kissner, Alina Oprea, Michael K Reiter, Dawn Song, and Ke Yang. Private Keyword-Based Push and Pull with Applications to Anonymous Communication. In Markus Jakobsson, Moti Yung, and Jianying Zhou, editors, *Applied Cryptography and Network Security: Second International Conference, ACNS 2004*, pages 16–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[151] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *NSDI '16*, pages 255–273, Santa Clara, CA, March 2016. USENIX Association.

[152] Adam Langley, Nagendra Modadugu, and Wan-Teh Chang. Overclocking SSL. https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html, 6 2010.

[153] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for voip systems. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 639–652, New York, NY, USA, 2015. ACM.

[154] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 303–314, New York, NY, USA, 2013. ACM.

[155] Taeho Lee, Christos Pappas, David Barrera, Pawel Szalachowski, and Adrian Perrig. Source accountability with domain-brokered privacy. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 345–358, New York, NY, USA, 2016. ACM.

[156] Peter Lepeska. Trusted proxy and the cost of bits. http://www.ietf.org/proceedings/90/slides/slides-90-httpbis-6.pdf, 7 2014.

[157] Vincent Liu, Seungyeop Han, Arvind Krishnamurthy, and Thomas Anderson. Tor instead of ip. In *HotNets '11*, page 14. ACM, 2011.

[158] Vincent Liu, Seungyeop Han, Arvind Krishnamurthy, and Thomas Anderson. An Internet Architecture Based on the Principle of Least Privilege. Technical Report UW-CSE-12-09-05, University of Washington, September 2012.

[159] S. Loreto, J. Mattsson, R. Skog, H. Spaak, G. Gus, D. Druta, and M. Hafeez. Explicit Trusted Proxy in HTTP/2.0. Internet-Draft draft-loreto-httpbis-trusted-proxy20-01, IETF Secretariat, February 2014.

[160] Yannis Mallios, Sudeep Modi, Aditya Agarwala, and Christina Johns. Persona: Network layer anonymity and accountability for next generation internet. In *SEC*, pages 410–420. Springer, 2009.

[161] Jeremy Martin, Travis Mayberry, Collin Donahue, Lucas Foppe, Lamont Brown, Chadwick Riggins, Erik C. Rye, and Dane Brown. A study of MAC address randomization in mobile devices and when it fails. *CoRR*, abs/1703.02874, 2017.

[162] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI '14*, pages 459–473, Seattle, WA, 2014. USENIX Association.

[163] Jonathan Mayer, Patrick Mutchler, and John C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences*, 113(20):5536–5541, 2016.

[164] D. McGrew, D. Wing, Y. Nir, and P. Gladstone. TLS Proxy Server Extension. Internet-Draft draft-mcgrew-tls-proxy-server-01, IETF Secretariat, July 2012.

[165] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*, page 10, 2013.

[166] Alfred Menezes and Berkant Ustaoglu. On reusing ephemeral keys in diffie-hellman key agreement protocols. *International Journal of Applied Cryptography*, 2(2):154–158, 2010.

[167] Ralph C. Merkle. Secure communications over insecure channels. *Commun. ACM*, 21(4):294–299, April 1978.

[168] D. Meyer, L. Zhang, and K. Fall. Report from the IAB Workshop on Routing and Addressing. RFC 4984 (Informational), September 2007.

[169] U. Moeller, L. Cottrell, P. Palfrader, and L. Sassaman. Mixmaster Protocol Version 2. Internet-Draft draft-sassaman-mixmaster-03, IETF Secretariat, December 2004.

[170] J. C. Mogul and G. Minshall. Rethinking the tcp nagle algorithm. *SIGCOMM Comput. Commun. Rev.*, 31(1):6–20, January 2001.

[171] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. RFC 4423 (Informational), May 2006.

[172] Chitra Muthukrishnan, Vern Paxson, Mark Allman, and Aditya Akella. Using strongly typed networking to architect for tussle. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 9:1–9:6, New York, NY, USA, 2010. ACM.

[173] Dalit Naor, Amir Shenhav, and Avishai Wool. Toward securing untrusted storage without public-key operations. StorageSS '05, pages 51–56, New York, NY, USA, 2005. ACM.

[174] Jad Naous, Michael Walfish, Antonio Nicolosi, David Mazières, Michael Miller, and Arun Seehra. Verifying and enforcing network paths with icing. CoNEXT '11, pages 30:1–30:12, New York, NY, USA, 2011. ACM.

[175] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The Cost of the "S" in HTTPS. In *CoNEXT '14*, pages 133–140, New York, NY, USA, 2014. ACM.

[176] David Naylor, Matthew K. Mukerjee, and Peter Steenkiste. Balancing accountability and privacy in the network. In *SIGCOMM '14*, pages 75–86, New York, NY, USA, 2014. ACM.

[177] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *SIGCOMM '15*, pages 199–212, New York, NY, USA, 2015. ACM.

[178] David Naylor and Peter Steenkiste. Do You Know Where Your Headers Are? Comparing the Privacy of Network Architectures with Share Count Analysis. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, pages 4:1–4:7, New York, NY, USA, 2015. ACM.

[179] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, CCS '01, pages 116–125, New York, NY, USA, 2001. ACM.

[180] Con Nikolouzakis. Encrypted traffic grows 40% post edward snowden nsa leak. http://www.sinefa.com/blog/encrypted-traffic-grows-post-edward-snowden-nsa-leak. Accessed: Jan. 2015.

[181] M. Nottingham. Problems with Proxies in HTTP. Internet-Draft draft-nottingham-http-proxy-problem-00, IETF Secretariat, October 2013.

[182] Danny O'Brien. Ten Steps You Can Take Right Now Against Internet Surveillance. https://www.eff.org/deeplinks/2013/10/ten-steps-against-surveillance, October 2013.

[183] Mark O'Neill, Scott Ruoti, Kent Seamons, and Daniel Zappala. Tls proxies: Friend or foe? In *IMC '16*, pages 551–557, New York, NY, USA, 2016. ACM.

[184] Fotios Papaodyssefs, Costas Iordanou, Jeremy Blackburn, Nikolaos Laoutaris, and Konstantina Papagiannaki. Web identity translator: Behavioral advertising and identity privacy with wit. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, pages 3:1–3:7, New York, NY, USA, 2015. ACM.

[185] Kihong Park and Heejo Lee. On the effectiveness of route-based packet filtering for distributed dos attack prevention in power-law internets. In *SIGCOMM CCR*, volume 31, pages

15–26. ACM, 2001.

[186] Bryan Parno, Zongwei Zhou, and Adrian Perrig. Using trustworthy host-based information in the network. In *Proceedings of the 7th ACM Workshop on Scalable Trusted Computing (STC)*, October 2012.

[187] R. Peon. Explicit Proxies for HTTP/2.0. Internet-Draft draft-rpeon-httpbis-exproxy-00, IETF Secretariat, June 2012.

[188] Andreas Pfitzmann and Marit Köhntopp. Anonymity, unobservability, and pseudeonymity — a proposal for terminology. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, pages 1–9, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[189] Andreas Pfitzmann and Michael Waidner. Networks without user observability. *Computers and Security*, 6(2):158–166, May 1987.

[190] Feng Qian, Subhabrata Sen, and Oliver Spatscheck. Characterizing resource usage for mobile web browsing. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 218–231, New York, NY, USA, 2014. ACM.

[191] Barath Raghavan, Tadayoshi Kohno, Alex C. Snoeren, and David Wetherall. Enlisting ISPs to improve online privacy: IP address mixing by default. In *PETS '09*, pages 143–163, 2009.

[192] M G Reed, P F Syverson, and D M Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 1998.

[193] Michael K Reiter and Aviel D Rubin. Crowds: Anonymity for web transactions. *TISSEC*, 1(1):66–92, 1998.

[194] S. Renfro. Secure browsing by default. http://goo.gl/B7U3jv, July 2013.

[195] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000. Updated by RFCs 5785, 7230.

[196] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-15, IETF Secretariat, August 2016.

[197] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-18, IETF Secretariat, October 2016.

[198] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012. Updated by RFCs 7507, 7905.

[199] Ruben Rios, Jorge Cuellar, and Javier Lopez. Robust Probabilistic Fake Packet Injection for Receiver-Location Privacy in WSN. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 163–180. Springer Berlin Heidelberg, 2012.

[200] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[201] Jody Sankey and Matthew Wright. Dovetail: Stronger Anonymity in Next-Generation Internet Routing. In Emiliano De Cristofaro and Steven J Murdoch, editors, *PETS 2014*, pages 283–303. Springer International Publishing, Cham, 2014.

[202] Liron Schiff and Stefan Schmid. PRI: Privacy Preserving Inspection of Encrypted Network Traffic. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 296–303. IEEE, 2016.

[203] Steve Schneider and Abraham Sidiropoulos. Csp and anonymity. In Elisa Bertino, Helmut Kurth, Giancarlo Martella, and Emilio Montolivo, editors, *Computer Security ÃćÂĂÂŤ ESORICS 96*, volume 1146 of *Lecture Notes in Computer Science*, pages 198–218. Springer Berlin Heidelberg, 1996.

[204] Edward J. Schwartz, David Brumley, and Jonathan M. McCune. A contractual anonymity system. In *NDSS '10*, 2010.

[205] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. NSDI'12, Berkeley, CA, USA, 2012. USENIX Association.

[206] Andrei Serjantov and George Danezis. Towards an information theoretic metric for anonymity. In Roger Dingledine and Paul Syverson, editors, *Privacy Enhancing Technologies*, volume 2482 of *Lecture Notes in Computer Science*, pages 41–53. Springer Berlin Heidelberg, 2003.

[207] Marianne Shaw. Leveraging good intentions to reduce unwanted network traffic. In *Proc. USENIX Steps to Reduce Unwanted Traffic on the Internet workshop*, page 8, 2006.

[208] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM '12*, pages 13–24, New York, NY, USA, 2012. ACM.

[209] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *SIGCOMM '15*, pages 213–226, New York, NY, USA, 2015. ACM.

[210] Rob Sherwood, Bobby Bhattacharjee, and Aravind Srinivasan. P5: A protocol for scalable anonymous communication. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 58–, Washington, DC, USA, 2002. IEEE Computer Society.

[211] Elaine Shi, Bryan Parno, Adrian Perrig, Yih-Chun Hu, and Bruce Maggs. FANFARE for the common flow. Technical Report CMU-CS-05-148, 2005.

[212] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-nfv: Securing nfv states by using sgx. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, SDN-NFV Security '16, pages 45–48, New York, NY, USA, 2016. ACM.

[213] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26-March 1, 2017*, 2017.

[214] Irina Shklovski and Nalini Kotamraju. Online contribution practices in countries that engage in internet blocking and censorship. *CHI'11*, pages 1109–1118, 2011.

[215] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 44–, Washington, DC, USA, 2000. IEEE Computer Society.

[216] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), January 2001.

[217] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. SIGCOMM '02, pages 73–86, New York, NY, USA, 2002. ACM.

[218] Nick Sullivan. Keyless SSL: The Nitty Gritty Technical Details. https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/. Accessed: September 2016.

[219] Paul F. Syverson and Stuart G. Stubblebine. Group principals and the formalization of anonymity. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I*, FM '99, pages 814–833, London, UK, UK, 1999. Springer-Verlag.

[220] The Chromium Projects. Spdy. http://www.chromium.org/spdy.

[221] Gergely Tóth and Zoltán Hornák. Measuring anonymity in a non-adaptive, real-time system. In David Martin and Andrei Serjantov, editors, *Privacy Enhancing Technologies*, volume 3424 of *Lecture Notes in Computer Science*, pages 226–241. Springer Berlin Heidelberg, 2005.

[222] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. *Network and Distributed System Security Symposium (NDSS)*, 2010.

[223] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 137–152, New York, NY, USA, 2015. ACM.

[224] Matteo Varvello, Jeremy Blackburn, David Naylor, and Konstantina Papagiannaki. EYEORG: a platform for crowdsourcing web quality of experience measurements. In *CoNEXT '16*, 2016.

[225] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 15–15, Berkeley, CA, USA, 2004. USENIX Association.

[226] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *NSDI 2013*, 2013.

[227] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How speedy is spdy? In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 387–399, Seattle, WA, 2014. USENIX Association.

[228] Nicholas Weaver, Christian Kreibich, Martin Dam, and Vern Paxson. Here be web proxies. In Michalis Faloutsos and Aleksandar Kuzmanovic, editors, *Passive and Active Measurement*, volume 8362 of *Lecture Notes in Computer Science*, pages 183–192. Springer International Publishing, 2014.

[229] Chester Wisniewski. Path and hipster iphone apps leak sensitive data without notification. https://nakedsecurity.sophos.com/2012/02/08/apple-mobile-apps-path-and-hipster-and-leak-sensitive-data-without-notification/. Accessed: May

2015.

[230] wolfSSL. wolfSSL with Intel SGX. https://software.intel.com/en-us/sgx-sdk/download. Accessed: June 2017.

[231] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182, Hollywood, CA, 2012. USENIX.

[232] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of caching strategies in modern cellular backhaul networks. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 319–332, New York, NY, USA, 2013. ACM.

[233] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson. Spot me if you can: Uncovering spoken phrases in encrypted voip conversations. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 35–49, May 2008.

[234] Charles V. Wright, Lucas Ballard, Fabian Monrose, and Gerald M. Masson. Language identification of encrypted voip traffic: Alejandra y roberto or alice and bob? In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 4:1–4:12, Berkeley, CA, USA, 2007. USENIX Association.

[235] Xing Xu, Yurong Jiang, Tobias Flach, Ethan Katz-Bassett, David Choffnes, and Ramesh Govindan. Investigating transparent web proxies in cellular networks. 2015.

[236] 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, 2004. Proceedings. 2004*, pages 130–143, May 2004.

[237] Xiaowei Yang, D. Wetherall, and T. Anderson. TVA: A DoS-limiting network architecture. *Networking, IEEE/ACM Transactions on*, 16(6):1267–1280, 2008.

[238] Zhiqiang Yang, Sheng Zhong, and Rebecca N. Wright. Privacy-preserving queries on encrypted data. In *Proceedings of the 11th European Conference on Research in Computer Security*, ESORICS'06, pages 479–495, Berlin, Heidelberg, 2006. Springer-Verlag.

[239] Bidi Ying, J.R. Gallardo, D. Makrakis, and H.T. Mouftah. Concealing of the Sink Location in WSNs by artificially homogenizing traffic intensity. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, pages 988–993, April 2011.

[240] Xin Zhang, Hsu-Chun Hsiao, G. Hasker, Haowen Chan, A. Perrig, and D.G. Andersen. SCION: Scalability, control, and isolation on next-generation networks. In *Security and Privacy 2011(SP), 2011 IEEE Symposium on*, pages 212–227, 2011.

[241] Xin Zhang, Zongwei Zhou, Geoff Hasker, Adrian Perrig, and Virgil Gligor. Network fault localization with small TCB. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, October 2011.

[242] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. CoNEXT '13, pages 97–108, New York, NY, USA, 2013. ACM.