PREVENTING ENCRYPTED TRAFFIC ANALYSIS

BY

NABÍL ADAM SCHEAR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Assistant Professor Nikita Borisov, Chair, Director of Research
Dr. Karen L. Bintz, Los Alamos National Laboratory
Assistant Professor Matthew Caesar
Professor Carl A. Gunter
Professor David M. Nicol

ABSTRACT

Many existing encrypted Internet protocols leak information through packet sizes and timing. Though seemingly innocuous, prior work has shown that such leakage can be used to recover part or all of the plaintext being encrypted. The prevalence of encrypted protocols as the underpinning of such critical services as e-commerce, remote login, and anonymity networks and the increasing feasibility of attacks on these services represent a considerable risk to communications security. Existing mechanisms for preventing traffic analysis focus on re-routing and padding. These prevention techniques have considerable resource and overhead requirements. Furthermore, padding is easily detectable and, in some cases, can introduce its own vulnerabilities.

To address these shortcomings, we propose embedding real traffic in synthetically generated encrypted cover traffic. Novel to our approach is our use of realistic network protocol behavior models to generate cover traffic. The observable traffic we generate also has the benefit of being indistinguishable from other *real* encrypted traffic further thwarting an adversary's ability to target attacks. In this dissertation, we introduce the design of a proxy system called TrafficMimic that implements realistic cover traffic tunneling and can be used alone or integrated with the Tor anonymity system. We describe the cover traffic generation process including the subtleties of implementing a secure traffic generator. We show that TrafficMimic cover traffic can fool a complex protocol classification attack with 91% of the accuracy of real traffic. TrafficMimic cover traffic is also not detected by a binary classification attack specifically designed to detect TrafficMimic.

We evaluate the performance of tunneling with independent cover traffic models and find that they are comparable, and, in some cases, more efficient than generic constant-rate defenses. We then use simulation and analytic modeling to understand the performance of cover traffic tunneling more deeply. We find that we can take measurements from real or simulated traffic with no tunneling and use them to estimate parameters for an accurate analytic model of the performance impact of cover traffic tunneling. Once validated, we use this model to better understand how delay, bandwidth, tunnel slowdown, and stability affect cover traffic tunneling.

Finally, we take the insights from our simulation study and develop several biasing techniques that we can use to match the cover traffic to the real traffic while simultaneously bounding external information leakage. We study these bias methods using simulation and evaluate their security using a Bayesian inference attack. We find that we can safely improve performance with biasing while preventing both traffic analysis and defense detection attacks. We then apply these biasing methods to the real TrafficMimic implementation and evaluate it on the Internet. We find that biasing can provide 3-5x improvement in bandwidth for bulk transfers and 2.5-9.5x speedup for Web browsing over tunneling without biasing.

for Melissa, Alina, and Zia

ACKNOWLEDGMENTS

I would like to thank and acknowledge the following (in no particular order):

- My friends and family for providing me with the patience and encouragement that I needed to get through this.
- My father for continually encouraging and believing in me throughout my scholastic career. Despite my occasional reservations and frustrations with school, he seemed to know that completing a Ph.D. was destined for me. Thank you for your persistence.
- My former and current research group at Illinois led by Nikita Borisov including Robin Snader, Shishir Nagarga, Amir Houmansadr, David Albreacht, and Prateek Mital. Working with you all has taught me more than any class.
- The sysnet group at UCSD for convincing me that doing graduate school was actually fun and not nearly as pointlessly grueling as undergrad had been.
- Colleagues at LANL who provided encouragement and help setting up experiments and servers. Also special thanks to those who provided much-needed proofreading: Curt Canada, David Cremer, Jeff Inman, and Dean Prichard.
- The unknown guy at DEF CON 14 who I met and spoke with in the Metasploit SkyBox. Your conversation with me about evading traffic statistics based intrusion detection planted the seed for this work.
- BJ Harshfield for providing some early feedback on this idea in 2006.

- Anthony, Danny, and various other Blackhat/DEF CON bigguns whose belief (some to this day others I have convinced otherwise) that one can just shove something into SSL or cryptcat and no one will ever be able to tell it from real traffic. This misconception motivated me to do better by creating TrafficMimic.

- Dean Prichard for providing help with S^3 and other advice when my math/statistics abilities failed me.

- Los Alamos National Lab ACS, CAIDA and the UNC Distributed Systems group for providing access to packet traces used in this work.

- Karen Bintz and Andy Erickson for help in getting funding through NCIX to continue this research at LANL.

- David Nicol for helping to solidify the soundness of this approach by creating an analytic model of tunneling and helping get my publication of this work started with the PADS conference.

- This work was funded in part by NSF CNS Grants CNS-0831488, CNS-0524695, and CNS-0627671.

- This work was funded in part by the ODNI National Counter-Intelligence Executive (NCIX) Counter Intelligence Integration Initiative Grant (CI-3).

This material is unclassified and released under LA-UR 11-01317 by the Los Alamos National Laboratory.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CDF	Cumulative Distribution Function
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IETF	Internet Engineering Task Force
IP	Internet Protocol
OSI	Open Systems Interconnection
PDF	Probability Density Function
SMTP	Simple Mail Transfer Protocol
SSL	Secure Sockets Layer
SSH	Secure Shell
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TM	TrafficMimic
VoIP	Voice over Internet Protocol
VPN	Virtual Private Network

CHAPTER 1

INTRODUCTION

As more sensitive information is transmitted over computer networks, there has been a steady increase in the deployment of encryption to protect data in-flight. Encryption is critical to e-commerce and has been key to the broad impact and success of the Internet. Myriad encrypted network protocols have emerged (e.g., [1–4]) that enable applications like encrypted Web browsing, VPNs, secure shells, and VoIP. Though encrypted protocols have been implemented at every layer of the OSI model, in this work we focus primarily on SSL/TLS[1], the most common application-layer end-to-end encrypted protocol used on the Internet.

Since the data payload of an encrypted protocol is protected by strong encryption, attackers use the information leaked by side channels to recover the contents or intent of the plaintext traffic. This type of attack, called traffic analysis, uses the sizes and timing of packets sent across the network. Though traffic analysis has been a well-studied spying technique for many decades, it has recently found applications in attacking Internet traffic.

To further motivate this problem, consider the following example of traffic analysis of encrypted Web browsing using the HTTPS protocol. Figure 1.1 depicts this scenario. The canonical activity of Web browsing is that a small request is followed by a larger response. The SSL tunnel created by HTTPS does not mask this behavior. An adversary (Alice) with appropriate access to monitor the network can see the encrypted payloads as they cross the network. She can also see the boundaries between requests and responses. Therefore, if Alice has access to the encrypted content on a Web site

[1]SSL was formally renamed TLS when it was standardized by the IETF. In this work, we use the term SSL to refer to the SSLv3/TLSv1 protocol in use today.

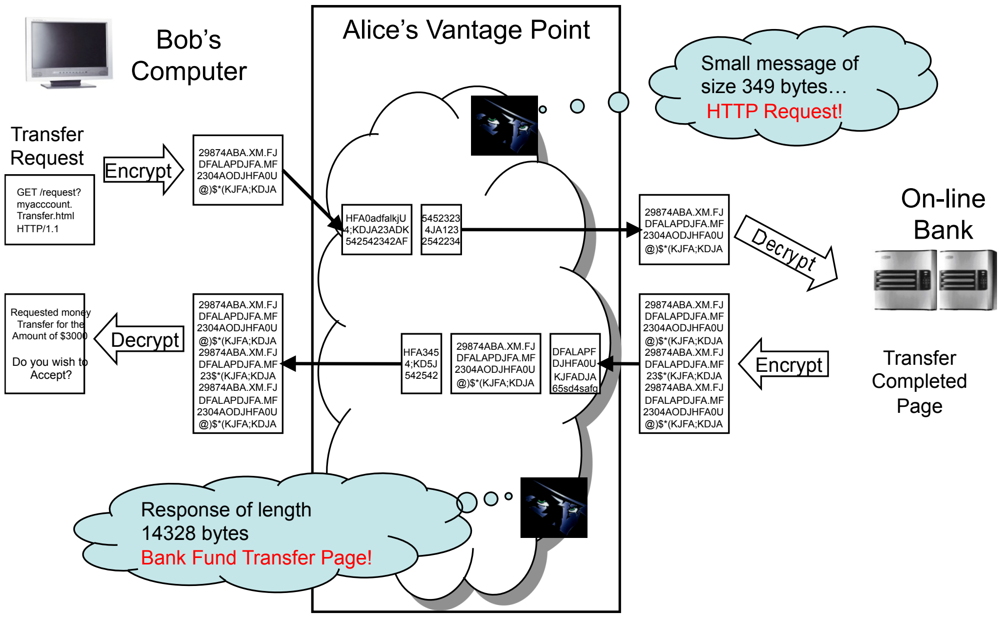

Bob's Computer
Alice's Vantage Point
Small message of size 349 bytes… HTTP Request!
Transfer Request
GET /request? myacccount. Transfer.html HTTP/1.1
Encrypt
On-line Bank
Decrypt
Requested money Transfer for the Amount of $3000 Do you wish to Accept?
Decrypt
Encrypt
Transfer Completed Page
Response of length 14328 bytes Bank Fund Transfer Page!

Figure 1.1: A simple example of encrypted traffic analysis of the HTTPS protocol

(for example, she and the victim (Bob) both have on-line banking accounts at the same bank) then Alice can catalog the size of each individual object on the site. Once she has done this, she can snoop Bob's traffic and reliably determine the pages he has visited without directly compromising any cryptographic keys, algorithms, or protocols. Alice can deduce, for example, that Bob transferred money because he has been to the *transfer-request* and the *transfer-completed* pages.

Existing traffic analysis attacks can recover a wide range of information from encrypted communications, e.g., Web page visits [5–7], typed passwords [8, 9], speech data [10, 11], and embedded protocols [12, 13]. Some users can tolerate such vulnerabilities; however, a growing number of applications (e.g., low-latency anonymity systems and VPNs) and users (e.g., whistle-blowers and people in oppressive regimes) need better protection from these attacks.

Existing techniques for preventing traffic analysis center on sending data with fixed intervals and/or with fixed payload sizes. Prior work has shown that such constant-rate techniques can result in considerable overhead [6, 10]. Furthermore, packets legitimately dropped during a constant-rate stream increase the effectiveness of flow linking attacks in mix systems [14]. Though the constant-rate defense *is effective* at reducing the information leakage that enables most traffic analysis attacks, is also clearly reveals that a user is employing countermeasures to evade traffic analysis. This may, in itself, result in unwanted attention and scrutiny. We call this related attack *defense detection*.

To resist *both* traffic analysis and defense detection attacks, we propose using realistic cover traffic tunnels to mask the observable behavior of the real traffic to be transmitted. Thus, the cover traffic model dictates when and how much data should be transmitted. Real data is queued until there is cover traffic available to send it. If there is no real data to send, the cover traffic is padded appropriately and sent. Strong encryption prevents the attacker from separating what portions of the network communications contain real data and which are padding.

In this dissertation, we present the design, implementation, and evaluation of a cover traffic tunneling system called TrafficMimic. The following statement summarizes the objectives of this dissertation:

> *Tunneling real data through realistic cover traffic models is a robust defense that provides balanced performance and security against powerful traffic analysis attacks.*

Existing encrypted protocols provide minimal traffic analysis resistance in exchange for better performance. This work provides users the option of changing this trade-off to favor more secure communications thereby balancing security and performance better than existing solutions.

The generality of the cover traffic model allows the user great flexibility in combating attacks. The degree of independence between the real traffic and the cover traffic drives how probable it is for an attacker to recover information from the session. It also influences the performance of the real traffic compared to its native performance without tunneling. Initially, we study cover traffic processes that are independent from the real traffic that they carry. We call these *independent* cover traffic models. We then show how to loosely couple the cover traffic process to the requirements of the real traffic while keeping attacker observable information leakage to a minimum. We call these *biased* cover traffic models.

We then show that using cover traffic models can robustly counter both traffic analysis and defense detection better than constant-rate techniques. Our defenses are robust because they correct the information leakages in current protocols and do not depend on specific attack implementation weaknesses. Since we assume the attacker's capabilities may range from those of a stealthy intruder to the network operator, we develop defenses that thwart even the most powerful attacks using the most advanced methods. Specifically, we target attacks that rely on the information leaked by common end-to-end encryption protocols currently used on the Internet (e.g., packet timing, payload sizes, etc).

To address these research challenges, we utilize simulation, analytic modeling, statistical machine learning, and full system implementations. Since this work has direct applicability to real privacy enhancing technologies, we ensure that the results of our research are focused on practicality. This includes developing and testing usable system implementations with quantifiable risks to attack and utilizing real network trace data to drive the learning and attack processes.

1.1 Structure of the Dissertation

The structure of this dissertation is as follows: The remainder of Chapter 1 covers the motivation and threat model for our work. This includes a review of recent traffic analysis attacks and their effectiveness. We then develop a threat model that favors the attacker in terms of access and resources.

In Chapter 2, we describe the design and implementation of our cover traffic tunneling system: TrafficMimic. We present methods for generating realistic cover traffic, borrowing from prior work on traffic generation from the simulation and modeling research community [15]. Since the quality of the cover traffic in our work is security-sensitive, we describe how existing traffic generators need to be modified to ensure they do not leak information. We then describe the system design and C++ implementation details of TrafficMimic. We also present a practical application of TrafficMimic by integrating cover traffic tunneling with Tor [16].

Chapter 3 contains an evaluation of the security and performance properties of our system implementation of TrafficMimic. We develop several protocol classification and anomaly detection attacks that are based, in part, on recent research in this area [12, 13, 17]. We show that TrafficMimic is able to fool our complex traffic classification attack 91% of the rate at which it works for real traffic. To address defense detection, we show that TrafficMimic cover traffic is indistinguishable from real traffic, even when attacked with a classifier trained with TrafficMimic models. We also show that realistic cover traffic can, in some cases, provide comparable performance to static constant-rate cover traffic. We then discuss performance problems that arise from mismatches between properties of the cover and real traffic.

In Chapter 4 we further examine the performance effects we observed in Chapter 3 using a simulation model and implementation in SSFNet [18]. We then develop a bi-directional analytic model for independent cover traffic tunneling and validate it using the results of our simulation study. This validated model allows us to understand performance properties of the real traffic like slowdown and the stability of cover/real traffic combinations.

We next investigate techniques for biasing the selection of cover traffic to better match the real traffic being tunneled in Chapter 5. We derive several statistical and algorithmic biasing functions and describe their implementation in a simulator. We use this simulator along with a Bayesian inference attack to compare the bias functions' security and performance. We study the effects of biasing on defense detection using the two sample Kolmogorov-Smirnov test. We then evaluate biased cover traffic models in our system implementation of TrafficMimic. We observed up to 5x improvement in bulk transfer performance and 2.5-9.5x improvement in Web site load time as compared to independent cover traffic. We also found that bidirectional overhead was also reduced using biasing. This study shows that we can safely enhance the real-world performance of TrafficMimic using biasing.

```
Num   Time Delta   From       To         TCP flags      TCP info
  1   0.000000     43620  >  https     [PSH, ACK]     Seq=1 Ack=1 Len=693
  2   0.037718     https  >  43620     [ACK]          Seq=1 Ack=694 Len=0
  3   0.467329     https  >  43620     [ACK]          Seq=1 Ack=694 Len=1448
  4   0.000123     https  >  43620     [ACK]          Seq=1449 Ack=694 Len=1448
  5   0.000023     43620  >  https     [ACK]          Seq=694 Ack=2897 Len=0
  6   0.000092     https  >  43620     [ACK]          Seq=2897 Ack=694 Len=1448
  7   0.000649     https  >  43620     [ACK]          Seq=4345 Ack=694 Len=1448
  8   0.000018     43620  >  https     [ACK]          Seq=694 Ack=5793 Len=0
  9   0.000091     https  >  43620     [ACK]          Seq=5793 Ack=694 Len=1448
 10   0.000068     https  >  43620     [PSH, ACK]     Seq=7241 Ack=694 Len=658
 11   0.000014     43620  >  https     [ACK]          Seq=694 Ack=7899 Len=0
```

Figure 1.2: Portion of a packet trace of HTTPS browsing taken from tshark [19]

We present a review of relevant related work in traffic analysis attacks, attack prevention, and network traffic generation in Chapter 6. We then conclude and describe future work in Chapter 7.

1.2 Traffic Analysis Threats

To establish effective techniques for preventing traffic analysis, we first examine the mechanics of precisely what information is currently leaked by encrypted protocols. We do so using an example taken from a portion of an HTTPS connection. We show the output in a format similar to that of Wireshark [19] in Figure 1.2.

The first information we can readily glean from this trace is that the *two hosts are communicating* over the HTTPS protocol. Unless being used in concert with a proxy, tunnel, or VPN service, current encrypted protocols do not directly obscure who is communicating. When used for a VPN, encrypted protocols may obscure the exact end-points of the communications since the IP headers are also encrypted. However, an attacker is still able to tell the end-points of the VPN (i.e., what organization hosts it). In our work, we do not prevent the attacker from determining the end points of the communications because existing work on anonymous relays and Internet blocking resistance can provide this.

The next thing we can observe from the above trace is the *timing of the packets*. Since we collected this trace at the client, we can observe round trip time of the first data packet as approximately 38ms. We can also see when the client and server send data. The next thing we can observe is the *length of the payloads in the packets*. We can tell that the client sent a total of 693 bytes and the server sent 7898 bytes in this portion of the trace.

Using the information we have observed so far, we can apply what we know of the normal operation of HTTP to reconstruct what is taking place. We see that a GET request was likely sent in packet 1. Using the timing of the packets, we deduce that the server processing time was approximately 467 ms. In a typical HTTP connection, the server sends back HTTP response headers and the content requested. We collected the response headers by crawling over one million pages from the Internet and found them to have a mean length of 212 bytes. Thus, we estimate the size of the object requested was approximately 7686 bytes (7898-212). After examining the corresponding plaintext session, we found our estimates from the encrypted session to be within close margins.

1.3 Related Work on Traffic Analysis Attacks

As evidenced by the preceding example, encrypted protocols can leak considerable amounts of information. Encrypted traffic analysis has been the subject of research since the advent of encrypted Internet protocols [20, 21]. To further understand this threat, we next present a taxonomy of traffic analysis against Internet encrypted protocols.

1.3.1 Anonymity Attacks

Anonymity systems are a common target for traffic analysis attacks. By hiding information about each user's identity, anonymity systems force attackers to use side channels to glean information about user identity. Raymond provides an overview of traffic analysis techniques against anonymity systems [22]. One such side channel attack is to correlate traffic patterns entering and leaving an anonymity system [23, 24]. Hintz et al. demonstrated a different type of attack on the SafeWeb system [25] that utilized a

HTTP protocol-specific attack (similar to those described in Section 1.3.4).

Zhu and Bettati developed another class of anonymity attacks they call flow separation [26]. They use blind source separation techniques to distinguish different flows inside a mix network. While their attack does not violate anonymity directly, their techniques make it easier to apply traffic analysis techniques on mixed traffic.

There are two approaches to using cover traffic in anonymity systems: link padding and end-to-end padding. Link padding ensures that a passive adversary observing traffic between the routers is unable to learn any information about the flows being carried. The use of realistic cover traffic in this scenario would help hide the fact that a router is forwarding anonymous traffic. Link padding, of course, does not provide a defense against compromised routers. End-to-end padding can be used to address this problem, but it is vulnerable to network perturbations, intentional or otherwise, that introduce distinct patterns in different flows [27]. An end-to-end *realistic* cover stream used by an anonymity system will likewise not help resist traffic analysis aimed to link anonymous flows, as each stream will have a distinct signature, but it will help protect from other attacks described below.

1.3.2 Stepping Stone Detection

Traffic analysis is also used to detect stepping stones [28], which are compromised computers that are used to relay malicious traffic to hide its true origin. The methodologies used are often similar to traffic analysis in anonymous communication networks [29, 30]. In this scenario, constant-rate cover traffic is likely to arouse suspicion. Donoho et al. [31] suggested constraining the cover traffic distribution to preserve a Pareto inter-arrival distribution and studied detection mechanisms that exploit this constraint, while Blum et al. analyzed using cover traffic following a Poisson process [32]. Realistic cover traffic would, of course, be the stealthiest approach to avoiding stepping stone detection.

1.3.3 Protocol Detection

Protocol detection is increasingly important both for security and for operational reasons. Intrusion detection systems with appropriate knowledge of the protocol may be better able to analyze traffic [33]. Also, it may be necessary to identify unknown protocols and subject them to further examination or a specialized routing policy (e.g., VoIP). Many protocol detection techniques are based upon examining the content of the traffic data [34].

The contents are obscured in encrypted protocols, so more sophisticated methods must be employed. Moore and Zuev use a Naïve Bayes classifier to classify individual flows using flow length, port numbers, and inter-flow time [17]. Wright et al. use hidden Markov models to identify encrypted and plaintext protocols using only packet timing and size [35]. They address some of the difficulties in utilizing so few features (time and size) in encrypted protocols, but still find that detection can be error-prone. They also propose methods for identifying traffic using aggregates of size information only [13]. They use the K-nearest-neighbor algorithm in conjunction with the Kullback-Leibler divergence to classify traffic aggregates collected in ten second intervals.

The work by Dhamankar and King used a large set of features (such as client/server traffic ratio, entropy, and standard deviation of request/response times) [12]. They also used a different classification mechanism by doing K-means clustering and then nearest neighbor using both Euclidean and Mahalanobis distance metrics. Their results focused on VoIP traffic (specifically Skype), but they were also successful at detecting other UDP protocols, such as NetBIOS, within small error margins.

While most cover traffic techniques are effective at preventing protocol detection, realistic cover traffic can be tuned so that a particular cover protocol is recognized. This is especially relevant since corporations and governments are now employing network anomaly detection systems (NADS) to track encrypted traffic usage. This technology can be used to enforce network security policies, such as preventing unauthorized VPN connections or catching malware command and control channels. Several commercial vendors offer NADS products including Securify, PacketMotion, Mazu Networks, and

Arbor Networks.

1.3.4 Protocol-Specific Attacks

A common attack against HTTPS is to identify the web site being browsed when the destination address is not available (e.g., when using an anonymous proxy) [36]. Initial approaches used individual object sizes to identify pages on the Internet [7] or correlate links across pages [37]. Later work added packet inter-arrival times to the prior metrics to identify which site a user was browsing [5]. The attack decreases in accuracy as the time between training the model and applying it to test traffic increases. Liberatore and Levine went on to evaluate several techniques for identifying encrypted HTTP streams [6]. For their techniques, they found a much smaller reduction in attack accuracy than prior work as the time between training and attack grew. Shi et al. propose another mechanism for identifying a site through a trace of encrypted browsing [38]. The novelty of their approach is to represent a browsing session as a vector of connection counts and data volumes transferred on each connection. They then use a support vector machine to classify the session vectors and differentiate Web sites. Herrmann et al. studied a Multinomial Naïve Bayes classifier on several different types of encrypted proxy systems. They also studied some of the difficulties associated with false positives and browser caching.

Monrose and Rubin discovered that by examining the keystroke timing behavior, they could extract the text being typed into a real-time system [8]. Song et al. used this idea to extract typed passwords (from stepping-stone connections) and commands from SSH sessions [9].

Because of its quality of service and bandwidth requirements, detecting VoIP is of interest to network operators. Indeed, this was the primary motivation for the work of Dhamankar and King [12]. Wright et al. have taken detection further to try to infer the language being spoken using only features of the encrypted packets being sent [11]. The variable bit-rate encoders used by many VoIP schemes leak information in packet sizes because the different sounds that make up speech are encoded differently. With a simple classifier they were able to achieve up to 90% accuracy for many languages. They also used hidden Markov models to detect some spoken phrases without the need to train on audio samples of the speaker [10].

Realistic cover traffic models can be used to destroy the specific features extracted by the above attacks while preserving the overall traffic model. For example, by using an HTTPS model to generate cover traffic for an HTTPS session, protocol identification will correctly classify the activity, but a website identification will be misled.

1.4 Threat Model

To establish the effectiveness of our prevention techniques, we use a threat model that is specific to preventing certain types of traffic analysis and favors the attacker in terms of resources and access. We focus on obscuring the timing, payload sizes, and behavior patterns of encrypted traffic. We are not able to reduce the amount of data to be sent, but we can obscure its semantics from the attacker. We assume the attacker may *passively* observe traffic at any point after it has been encrypted and may do so with high-precision timing. We assume that the attacker cannot directly compromise any encryption key or algorithm and must focus on side-channel analysis. We also consider the adversary to have considerable resources at his/her disposal and to have no need to be stealthy. Our system does not obscure *who* is communicating or whether two parties are communicating at all. However, when used with an anonymity network, proxy, or covert channel, which can prevent flow linking, our work is capable of minimizing the information an attacker can recover from any type of known traffic analysis attack.

Since protocol classification may be required for an attacker to know which protocol-specific attacks to carry out, we focus on preventing protocol classification attacks. Protocol classification is also central to conducting and defending against the defense detection attack. Thus, we use a supervised learning protocol identification traffic analysis attack. We assume that the attacker has access to generic network traces with which he/she trains the protocol identification attack. The attacker then uses this trained classifier against specific target networks and hosts. We also use an unsupervised-learning method to identify protocol anomalies (i.e., constant-rate traffic flowing over a port which usually contains a real protocol). We describe the design and implementation of these attacks in more detail in Section 3.1.

Finally, we note that in addition to malicious attacks on user privacy, there are many scenarios where traffic analysis plays a benevolent role in network security. For example, network operators may use traffic analysis to perform network policy enforcement and intrusion detection thereby improving security for their users. Network policies that forbid unauthorized outbound virtual private networks can protect internal resources from attack and help mitigate exfiltration of sensitive information. To alleviate the confusion over the ambiguous motivations of those doing traffic analysis, we assume that those conducting the attack are the adversaries.

CHAPTER 2

TRAFFICMIMIC DESIGN

The primary purpose of TrafficMimic is to prevent traffic analysis attacks, like those discussed in Section 1.2. Since these attacks use the information (e.g., packet sizes and timing) that SSL and other end-to-end encryption protocols leak, we must leak as little information as possible from TrafficMimic. The most basic approach to stop information leakage is to map the real traffic onto a simple flat protocol model like constant-rate. Indeed, this method has been used with success in mitigating traffic analysis attacks in prior work [6, 10, 25, 37]. However, using such simple protocol models appears anomalous and is easily detectable by the adversary. To ensure responsive and efficient communications, constant-rate traffic must be emitted at the maximum desired rate all the time, resulting in considerable overhead. Furthermore, these methods are difficult to tune due to their mismatch with real Internet protocols (and excessive tuning may result in information leakage of its own).

To best address the problems with existing traffic analysis resistance, we use *realistic protocol behavior models* to generate cover traffic rather than constant-rate traffic. Using realistic models allows a user to ensure the attacker sees the protocol behaviors that are expected of the port, user, or location, thwarting defense detection. Realistic models can also provide performance advantages, especially with respect to excess padding overhead.

Realistic cover traffic models can either be independent or dependent of the real traffic flowing through them. When using an independent cover traffic model, TrafficMimic provides the same strong level of traffic analysis resistance as constant-rate models while simultaneously preventing defense detection. In Chapter 5 we study how to relax independent models while quantifying the information loss.

The overhead incurred by using an independent realistic model varies greatly depending on the properties of the cover traffic and the requirements of the real traffic. However, similar to how the simple models may be tuned via parameters, the user may select an independent model based upon his/her requirements for tunneled real traffic. A good deal of automatic tuning/fitting takes place when using the same model for cover traffic as the real traffic. For example, a user may tunnel his/her real Web browsing session over independent Web browsing cover traffic.

2.1 Secure Traffic Generation

So far, we have espoused that using realistic cover traffic has certain security benefits. However, this assumption is highly dependent on the quality of the generated cover traffic (i.e., how realistic is it?). Studying relevant work in realistic traffic generation from the simulation and performance analysis communities revealed that generating truly realistic traffic is complex and highly dependent on the beholder's idea of what features are important to realism. For example, it is often important for traffic generators used for network algorithm testing to properly emulate network link congestion [15] or for traffic generators used to test Web servers to properly exercise the caching mechanism [39].

Our goals in using realistic cover traffic are i) to minimize information leaks from the real traffic and ii) to ensure that the synthetic traffic of a given protocol is indistinguishable from actual traffic of that protocol. Since our application of traffic generation is security-sensitive due to defense detection attacks, we found that we needed a different set of requirements than those of previous work. We call this set of requirements and goals *secure traffic generation*.

We consider secure traffic generation to have two phases: learning and playback. Secure traffic generation shares many requirements for both learning and playback with existing traffic generators used for a variety of purposes. These include the requirement for realistic modeling of think time, objects passing over the network, and connections.

We next describe how secure traffic generation differs from existing work on traffic generation. We describe each of the requirements and the rationale behind them below.

1. **Network Agnostic and Responsive** We do not consider link or network properties to be a part of a secure traffic generation model because we want it to be responsive to actual network conditions when it is used for tunneling. Indeed, improperly responding to network conditions could enable a defense detection attack. Thus, when extracting information from network traces, we must be careful to focus on features that are salient to the protocol's behavior and minimize the effects any particular network has on learning secure protocol models. We generate protocol behavior models at the application layer so that they can actively react to congestion and load dynamically. We believe that models generated only from packet-level information (e.g., inter-packet timing) or that are played back using "open-loop" UDP traffic generators (e.g., Harpoon [40]) are not suitable for secure traffic generation.

2. **Quality/Type of Training** Secure traffic generation is only as good as its input data. Thus, it is important to ensure the training data is free of significant packet loss, inconsistencies, and route asymmetry. It is also important that the training data be recent to ensure that it will thwart defense detection attacks. Our experience with several freely-available anonymized packet traces used in this work indicate that, while suitable for supporting our work, these traces fall short for real use in an adversarial environment due to several of these quality issues. Lastly, since secure traffic generation involves a learning phase (in our design using a network trace), care should be taken to mitigate training-based attacks, which attempt to skew the learned model for the attacker's purposes. Others have studied this idea in the context of evading intrusion detection [41,42], and we find it similarly applicable to secure traffic generation.

3. **Synchronized Traffic Generation between Hosts** Differing from the previous two requirements, which focus on the learning phase, this requirement is relevant instead to the playback of secure traffic models. Since secure traffic models

Table 2.1: Features collected by Swing [15]

InterConn	Time between connections
numpairs	Number of req/resp *exchanges* per connection
req/resp	The size of individual requests and responses
reqthink	Wait time between exchanges on a connection

are *bidirectional*, we find that it is important that we use a *single synchronized* generator to create a cover traffic tunnel. The inconsistencies and anomalies that result from having two independent (or even semi-independent) traffic generators at either end of a secure tunnel can enable several defense detection attacks. Thus, secure traffic generation must ensure statistical, aggregate, and very specific heuristic properties of the modeled protocol.

To build the learning phase of our secure traffic generator we make use of structural models of observable protocol/network behavior. Structural models are a common technique used for simulation and performance testing of networks [43]. Structural models attempt to mimic the real interactions between the layers that generate network traffic (e.g., network, protocol, application, user). We use Swing, a recent network traffic generator [15], to create realistic protocol behavior models. Swing uses a structural model that includes user, session, connection, packet, and network sub-models to generate realistic traffic for network emulations. It does so by analyzing dumped network trace files and extracting empirical CDFs of structural features. Table 2.1 gives the features from Swing we used and their descriptions.

We discard the network-level details of the Swing models to ensure that they remain network agnostic. Some network timing attributes still leak into the structural models we use. Since, network-specific delays are likely to be small compared to both user and session timing, this is effect is minimal. However, some application layer timing information (e.g., reqthink) may be fine-grained enough to be affected by network-specific delays.

At first, we underestimated the effect that the quality of the input data to Swing would have on its ability reliably perform secure traffic generation. Network traces collected from high-bandwidth links, like those we use in this work, are often very difficult to process because of packet loss, routing anomalies, network scans, and end-device TCP stack implementation differences. Since Swing aggregates traffic properties across connections of a single port it observes in a trace, we found that the models it produces include data from malformed connections. Furthermore, Swing sometimes mis-handled TCP stream reassembly (e.g., it counts TCP phantom bytes as real transmissions) and was unable process to non-Ethernet link layer traces. While these problems have minimal impact on Swing in its primary purpose as a performance evaluation tool, we need to carefully sanitize input traces to preserve the security of the traffic generation process.

We developed a set of trace sanitization tools, which perform the following tasks:

- Remove connections where only one direction was observed. While these connections halves have some valuable information, the loss request/response correlation significantly skews Swing models.

- Remove connections with significant packet loss. These connections stress the end-device TCP stack and often expose non-fatal anomalies that are hard for Swing to properly reconstruct.

- Remove connections with only a few bytes transmitted. These connections are often result of network scans and do not contain enough real information to warrant retention.

- Reconstruct link-layer headers on which Swing depends. This ensures that Swing can track the direction of packets and correlate activities in both directions.

This preprocessing ensures that we only generate models from a set of *cleansed* connections that can be safely used with Swing.

We ensure synchronized playback of the models extracted by Swing using asynchronous generation threads and a lightweight inter-node communication mechanism for specifying generated traffic. We describe this system and the operation of the TrafficMimic tunnel next.

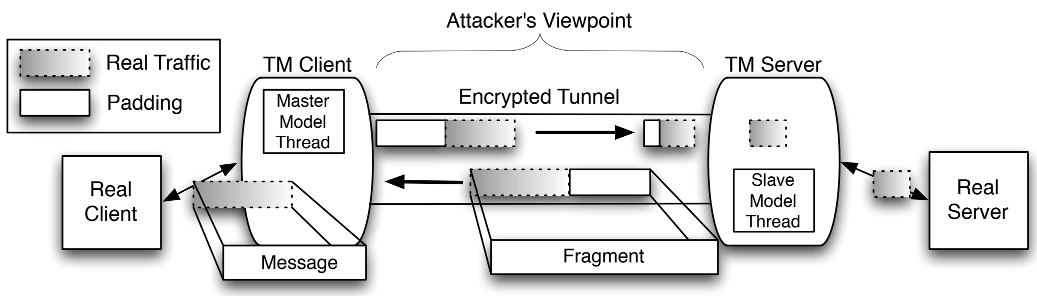


Figure 2.1: Tunneling real traffic through model-based cover traffic using TrafficMimic

2.2 Playback and Tunneling

TrafficMimic is a cover traffic playback system that incorporates a tunnel of real user traffic inside cover traffic. TrafficMimic is a TCP connection forwarding proxy service (it can also support UDP traffic over TCP cover traffic). TrafficMimic accepts connections via local/remote port forwarding, SOCKS, or HTTP CONNECT protocols and forwards all data sent and received over the input port across an encrypted tunnel.

To use TrafficMimic, the user must have TrafficMimic nodes at both end points of the communication. The remote TrafficMimic node removes the added control and padding information and delivers the decrypted data to the destination; likewise, data from the destination is embedded in cover traffic, encrypted, and padded at the remote node and decrypted by the local TrafficMimic node. As outlined in the threat model, we assume that the attacker can only see the encrypted traffic between the TrafficMimic hosts. Figure 2.1 shows the architecture of TrafficMimic.

Both endpoints of a TrafficMimic tunnel share the same design. We refer to the node accepting connections to forward as the *tm-client* and the end point connecting to destination hosts as the *tm-server*. To ensure synchronized cover traffic generation, a single node controls the generation of cover traffic for both nodes. We call this node the *master* and the other node the *slave*. The master may be at the tm-server *or* the tm-client. Since the master controls all cover traffic generation, the bidirectional cover traffic model can incorporate synchronized actions and predefined sub-sequences that would not be possible with two independent cover traffic generators at the tunnel end-points.

All cover traffic generation is performed in terms of *traffic request*s. A traffic request is a specification for what cover traffic TrafficMimic nodes will generate on a tunnel. A traffic request contains four elements: a type, a unique ID, a size, and a time. The type dictates whether the request is to send data or open/close/reconnect a connection. The size parameter specifies the number of bytes to send and the relative time parameter tells the system when to perform the requested action. The unique ID allows the master and slave to synchronize.

Each TrafficMimic node contains an asynchronous thread that handles the generation and processing of cover traffic. To create cover traffic, the master model thread uses Swing features to create *model request*s, which consist of a traffic request to be sent to the slave and a *model response*, which contains zero or more traffic requests that the slave should send back. TrafficMimic creates an appropriately sized output buffer according to the model request. TrafficMimic then embeds the model response and any real user data to be sent across the tunnel into the buffer. If additional space still remains, TrafficMimic pads it to the model-specified length. This buffer is then sent to the slave over an SSL-encrypted channel.

Upon receiving model responses from the master, the slave sleeps as specified and instructs its core network event loop to connect/disconnect the tunnel and to send data at the appropriate times. The slave uses the same process for merging control traffic, padding, and real user data as the master to create a bidirectional channel.

To allow the master initiated cover traffic to be synchronized with the data the slave sends back, the master model thread assigns a unique ID to each traffic request it generates. When the slave sends data back to the master, it includes the ID of the traffic request that it received to generate that data. This allows the master to wait for the responses from the slave to be sent back before proceeding with generating additional cover traffic.

Since master model threads are asynchronous from the rest of the proxy system and contain built-in synchronization, they are straightforward to write. Consider the simple model thread in Figure 2.2. Each call to *sendModelRequest* blocks until the associated

Figure 2.2: Psuedocode showing the implementation of a simple model thread

```
while True {
        // instruct master to create a new cover connection
        t1 = new TrafficRequest(CONNECT)
        sendModelRequest(new ModelRequest(t1))

        //  instruct master to send 500 bytes immediately
        t2 = new TrafficRequest(SEND_DATA, 500)
        m = new ModelRequest(t2)

        // create a container for slave responses
        resp = new ModelResponse()
        // instruct slave to send 1000 bytes after 20ms
        resp.add(new TrafficRequest(SEND_DATA, 1000, 20))
        // instruct slave to expect to be disconnected from the master
        resp.add(new TrafficRequest(WAIT_DISCONNECT))

        // associate the responses with the master request
        m.setResponse(resp)

        // send the master request and 2 slave responses
        sendModelRequest(m)

        // Disconnect cover connection
        t3 = new TrafficRequest(DISCONNECT)
        sendModelRequest(new ModelRequest(t3))

        // pause before beginning a new cover connection
        sleep(1)
}
```

action is complete. Furthermore, the model thread may simply sleep to emulate think time rather than requiring more complex synchronization with the main event loop. Thus, the master model thread is very general and easy to program for a wide variety of protocols while avoiding complex synchronization problems.

2.3 Implementation

Our TrafficMimic implementation consists of approximately 9,000 source lines of C++. TrafficMimic shares the configuration options and syntax for port forwarding, HTTP Connect, and SOCKS 4/4a/5 proxy servers with the OpenSSH client to ease adoption [44]. Users can integrate TrafficMimic with many standard applications that natively support proxies (e.g., most Web browsers, instant messengers, IRC clients, etc...) or with applications that do not natively support proxies using transparent solutions like proxychains, tsocks, or WinGate.

TrafficMimic uses two object interfaces for managing data: *messages* and *fragments*. Both are tied to a lightweight serialization interface, which allows objects to be converted into a heavily-packed byte array and efficiently moved across network sockets and IPC pipes. Messages are high-level objects that implement the functionality of TrafficMimic. They support sending real data through the tunnel, sending padding, resolving addresses, and connecting remote/local hosts using the TrafficMimic forwarding protocol.

Messages are divided into *fragments* when they are sent over the tunnel. This allows messages to be arbitrarily split across different transmissions and even across different TCP connections. Each fragment contains a very small header (as small as 2 bytes using varint integers [45]), model response data, and some portion of a message. We always encode model response data into each fragment to ensure that cover traffic generation continues even in cover traffic models with very small packets. The fragment serialization interface allows fragments as small as 8 bytes. Since encrypted block ciphers that are part of SSL and other encrypted protocols use 8 byte or greater block sizes, we can precisely mimic most encrypted traffic with TrafficMimic.

TrafficMimic uses an event loop powered by the `select` system call to monitor all in-use sockets. As stated earlier, TrafficMimic uses asynchronous model threads (implemented using pthreads) to govern traffic generation at the master and slave. These threads utilize IPC pipes and the same serialization interface used for external connections to asynchronously communicate traffic requests to the main network event loop. The main event loop communicates back to the model threads (for full traffic generation synchronization) with blocking feedback queues.

2.3.1 Tor Integration

The Tor project is an excellent candidate for integration with TrafficMimic because of its goal of traffic analysis resistance and anonymity [16]. Tor provides traffic analysis resistance by forwarding traffic through a set of anonymous relays. However Tor is vulnerable to both traffic analysis and defense detections that TrafficMimic mitigates.

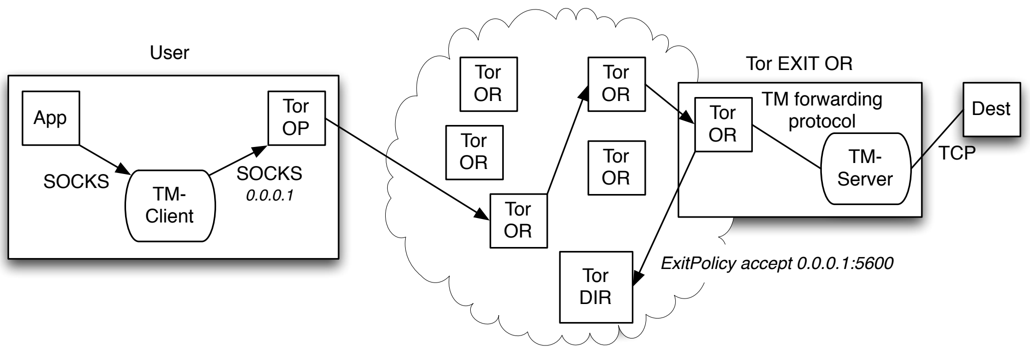


Figure 2.3: Using TrafficMimic end-to-end with Tor

It is especially vulnerable on the link between the Tor user and the first relay in the Tor network (first hop link). An attacker can reconstruct some information about a user's activities by monitoring this link (e.g., Web site identification [46]). Furthermore, an attacker can locate and block Tor traffic using a defense detection attack on this or any Tor link because of the unique SSL network fingerprint of Tor traffic [47].

Since TrafficMimic uses standard proxy initiation mechanisms with which Tor is compatible, we are able to integrate TrafficMimic into the Tor anonymity network in several ways. First, we can run TrafficMimic end-to-end over the Tor network, preventing first-hop traffic analysis attacks and making flow linking attacks by a global passive adversary considerably more difficult. Second, we can use TrafficMimic to protect individual Tor links between relays and users, preventing identification and blocking of Tor.

The usage model for using TrafficMimic end-to-end with Tor is as follows. The user instructs TrafficMimic to cover his/her protocol behaviors with realistic cover traffic before it enters the Tor network. The TrafficMimic covered data then traverses the Tor network eventually arriving at an exit-enabled Onion Router (OR). TrafficMimic runs locally on this OR and performs the final connection to the destination. To support this type of integration, we enhanced TrafficMimic to support the SOCKS protocol as a client as well as a server. Figure 2.3 shows the components of Tor and TrafficMimic and how they interconnect.

To help users who wish to use TrafficMimic with Tor, we add a special TrafficMimic exit policy that is propagated through the Tor directory. We do this by making a small

(3 line) change to the code of each OR that also runs a tm-server. We use an unallocated IP address (e.g., 0.0.0.1) as a placeholder in the exit policy for connecting to the TrafficMimic node. The OR code change maps this address to localhost to ensure that users can talk to the tm-server. Since this unallocated IP address will propagate through the Tor directory, users may connect to Tor using TrafficMimic without the need to change their Tor OP software, intermediate ORs, or the Tor directory, providing a path for incremental deployment.

There is some potential benefit to removing cover traffic in the middle of a Tor path, rather than at the last hop, since it would make traffic analysis at the entry and exit nodes (rather than by an external observer) more difficult. However, this would require a more significant modification to the Tor architecture.

The second integration protects individual SSL connections used by Tor. This prevents both traffic analysis and defense detection attacks for any link forwarded over TrafficMimic. Consider the example of a user protecting his/her first-hop link to Tor (Figure 2.4). The user configures his/her OP to connect to the local tm-client using the `HTTPSProxy` Tor configuration option (`ReachableAddresses` and `FacistFirewall` might be also necessary). The TrafficMimic-enabled entry guard OR should advertise port *n* for relay connections using the `ORPort` option while actually configuring the relay to listen on a different port *m* using `ORListenAddress`. The tm-server should listen on port *n* and forward incoming connections to localhost:*m*. This process will transparently inject TrafficMimic protection into any Tor link without changes to the directory service. To address some blocking-resistance problems, Tor bridge nodes (unpublished one-hop relays into the Tor network) can also use this technique.

2.4 Conclusions

In this chapter, we have presented the design and implementation of TrafficMimic, a system to resist traffic analysis by generating realistic cover traffic. TrafficMimic is able to both hide details about the traffic generated by users and also prevent defense

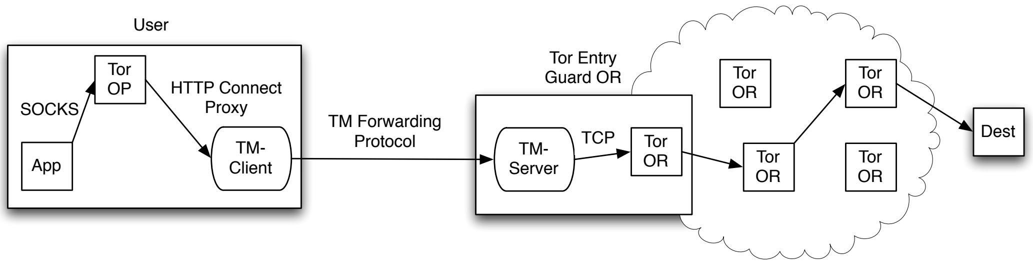


Figure 2.4: Using TrafficMimic to protect individual Tor links from detection and traffic analysis

detection by using realistic traffic models that do not appear anomalous. We discussed the requirements and our implementation of secure traffic generation using Swing. We also present the system architecture and implementation details of TrafficMimic that support easy-to-write traffic generation threads and careful synchronization. Lastly, we presented several usable integrations of TrafficMimic into the Tor anonymity network that improve its resistance to both traffic analysis and blocking attacks.

CHAPTER 3

INDEPENDENT COVER TRAFFIC EVALUATION

We evaluate TrafficMimic in two aspects: its security properties and the performance of using it for tunneling real traffic. We use three sets of network traces to train models and test our classification attacks. We use data from the CAIDA Anonymized 2009 Internet Traces Dataset [48] (January and February hour-long traces from the equinix-chicago monitor). Since we were unable to reconstruct a sufficient number of bidirectional SSH connections from these CAIDA traces, we also used several hour-long traces collected by the University of North Carolina on their campus-wide border link in April 2003 [49]. As an independent validation set, we also use one hour of traffic collected from Los Alamos National Laboratory's (LANL) border gateway link in Sept 2009.

We use several wide-area links to evaluate both security and performance (see Table 3.1). We run TrafficMimic nodes on the endpoints of these links and evaluate the tunnel traffic between them. While not fully representative of the Internet, these links vary enough to prove the effectiveness of our approach and its resistance to attack. In future work, we could evaluate TrafficMimic with more wide-area links by using PlanetLab [50].

Since Tor is likely to make protocol classification and other attacks more difficult [46], the results we present are a best case for the attacker. Tor likewise has a considerable impact on performance [51] that we plan to investigate in future work.

3.1 Traffic Classification Attack

To evaluate our traffic analysis resistance mechanisms we build upon existing attack methods [12, 13, 17]. In this section, we describe the design of our classification attack

Table 3.1: Wide-area test links

Label	**Client**	**Server**	**RTT**
MN-UK	Montreal	London	85ms
IL-CA	Illinois	California	63ms

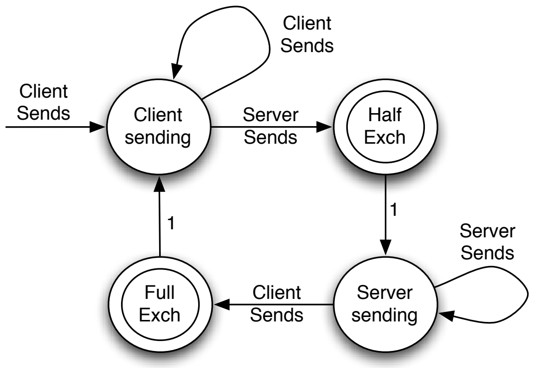
Table 3.2: Features extracted by the trace analysis system for use in our classification attack

Bytes Sent	Bytes Received
Mean Sent Pkt IAT	Mean Received Pkt IAT
Variance Sent Pkt IAT	Variance Received Pkt IAT
Number of Exchanges	Inter-Exchange Time
Connection Duration	

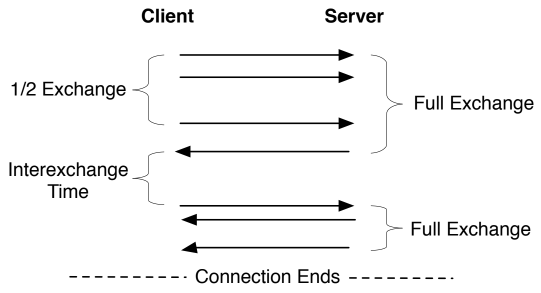
and its performance classifying real traffic. We later use this attack to validate secure traffic generation in Section 3.2.

The goal of the classification attack is to label a single connection of an unknown encrypted protocol with its plaintext protocol identity. Conservatively, we assume that we cannot use *any* information from the payload. Our classifier employs supervised learning and the weighted K nearest neighbor (K-NN) algorithm. For each example in the test set, our implementation calculates the Euclidean distance to each training example finding the K nearest neighbors. We then weight the neighbors according to their distance and classify each test sample based on the class with the highest weight. In our experiments, we found that values of K greater than 3 did not improve and in some cases reduced the overall accuracy of our classifier. Thus, we use $K = 3$ in all the following results.

We selected some standard features from the work Dhamankar and King [12] and Moore and Zuev [17]. We augmented these features with several features that showed promise for being salient in detecting differences between protocols (see Table 3.2 for the full list). The first of such new features relate to protocol *exchanges*. We define an exchange as a subsequence of network communications within a single connection where the initiator and responder both send data (i.e., one or more packets with non-zero payloads). This is similar to the concept of *pairs* in the work of Vishwanath and



(a) State machine for counting exchanges.



(b) Example showing two exchanges. Arrows depict packets going between the server and client.

Figure 3.1: Exchanges

Vahdat [15]. To reconstruct exchanges, we merge together all data packets that flow in one direction as one half of an exchange. When a data packet is sent back by the other peer, the second half of the exchange begins. This process repeats as the flow of data packets alternates between the initiator and responder. Though they do not cover all possible protocol interactions, exchanges capture the great majority of common protocol idioms. Refer to Figure 3.1 for a diagram of the exchange state machine and an example of how to determine exchange boundaries. We also use the total duration of the connection as a feature. In some cases, the length of our hour-long traces artificially limited this feature, though we still found it to improve classification performance. We experimented with various types of individual packet size related features, but we found that they were too noisy to improve the attack.

To extract relevant features from each TCP connection, we process packet traces using libpcap and perform TCP stream reassembly to avoid errors introduced by lost, delayed, and reordered packets. We use standard techniques to identify and in some cases infer when TCP connections are established [15]. We consider the initiator of the TCP connection (the party who sends the initial SYN packet) to be the sender and all data sent back to the initiator from the responder to be received data. For example, in a connection where a browser connects to a Web server, the data *sent* by the initiator (browser) is the HTTP request and the data *received* from the responder (Web server) is the HTML or other Web content requested. Although our trace analysis system does not support IP fragmentation, we found comparably few instances of fragmentation in the Internet traces we analyzed.

To support the scale of our input traces (on the order of 100 million packets), we created a custom feature extraction tool. By using careful resource control on the hash tables and lists we used to track connection status, our Java-based analysis system is able to process around one hundred thousand packet headers per second.

Since these features have vastly different units (e.g., seconds vs. bytes), we use z-scoring to standardize the values to a common range. To compute the z-score, we estimate the population standard deviation and mean using sampling from the training set. We used matrix conditioning to evaluate our choice of features. We use the matrix condition number for a normal matrix A given by

$$c_e = \frac{\lambda_{max}(A)}{\lambda_{min}(A)} \tag{3.1}$$

where λ_{min} and λ_{max} are the smallest and largest eigenvalues of A respectively. We consider a matrix with a c_e value roughly similar to its number of features to be well conditioned. We also used the trace of the inverse correlation matrix of A to identify redundant features. Using the full set of features from Table 3.2, we found the combined CAIDA and UNC training data to have a favorable c_e equal to 22.8.

3.1.1 Validation

We tested our classifier using cross-validation by training and testing data from the same network link at different times. We classified connections collected from CAIDA in February to those collected from January and from UNC on April 20 to April 29. To make the attack more realistic, we also tested the performance of our traffic classifier by training on traffic from one network (from CAIDA and UNC) and applying it against data collected from an entirely different network (from LANL). This approach differs from that taken by previous traffic identification work that used cross-validation of a single network link. Indeed, such generalization should reduce the accuracy and fidelity

Table 3.3: Results of traffic classification using K-NN classifier

Protocol	**Accuracy**	**F-score**
SSH	95.1%	0.886
STMP	88.7%	0.863
HTTPS	90.5%	0.874

of the techniques. Despite this, we feel that this threat model is most appropriate for an attacker wishing to perform wide-scale traffic analysis. Furthermore, we have designed our classifier such that it captures and compares features of the protocol's behavior and minimal information that are side effects of the network. We are the first to evaluate a generic traffic classification attack in this manner.

We focus on three common Internet protocols: HTTPS, SMTP, and SSH. Each represents a varying degree of interactivity, bulk transfer, and other protocol variations. As in previous work [13, 34], we use known port numbers to label our training and test sets. While this labeling mechanism is not strictly guaranteed to be correct (as our work demonstrates), we have achieved high classification performance in testing, minimizing the concern of mislabeling. Since the protocols we wished to analyze did not occur in the traces with equal probability, we employed random sampling to construct training and test sets for our classifier that had comparable numbers of connections from each protocol.[1]

We show the results of the single-link cross validation test in Table 3.3. The overall accuracy of the cross-validation CAIDA test of was greater than 91% and compares favorably with existing work using similar methods [12, 13, 17]. Since accuracy does not entirely reflect the performance of the classifier with respect to false positives and false negatives, we also report the F-score. The F-score is the harmonic mean of the precision and recall of the classifier and captures how many samples are missed and how many are incorrectly labeled simultaneously. We found the F-scores to be very good (>0.86) in this test.

Using the independent LANL trace, we found that the accuracy and F-score of our

[1]An attacker with a non-uniform prior probability of protocols carried by encrypted traffic would need to alter this methodology.

Table 3.4: Classification attack results validated with LANL data

Protocol	Accuracy	F-score
SSH	92.3%	0.78
SMTP	85.2%	0.863
HTTPS	82.0%	0.791

classifier suffered only ∼10% compared to results from the cross-validation experiment. We show the detailed results from this test in Table 3.4 (overall accuracy 80.1%). We also noted that the network conditions in our test traces are considerably different (CAIDA: recent inter-ISP, UNC: university border link from 6 years ago, and LANL: recent border link to a large research institution). Thus, the classifier still performs correctly in this generic test the vast majority of the time, even with considerable network differences exhibited in the traces.

3.2 Security Results

In this section we evaluate the realism of TrafficMimic using the classifier described above. We use two attack scenarios to evaluate our cover traffic. The first is a generic protocol classification attack and the second is specifically designed to detect TrafficMimic.

3.2.1 Protocol Classification Attack

We first evaluate the realism of TrafficMimic using a generic protocol identification attack. The goal of the attacker is to identify the underlying protocol running over or using encrypted transport. As described before, this type of attack is the first step for an attacker to target specific users or applications. Indeed, it can be considered a precursor to more valuable attacks like website fingerprinting or real time session keystroke attacks.

To construct the experiment, we use Swing to generate traffic parameters for HTTPS, SMTP, and SSH protocols learned from the Feb CAIDA 2009 and UNC April 29, 2003

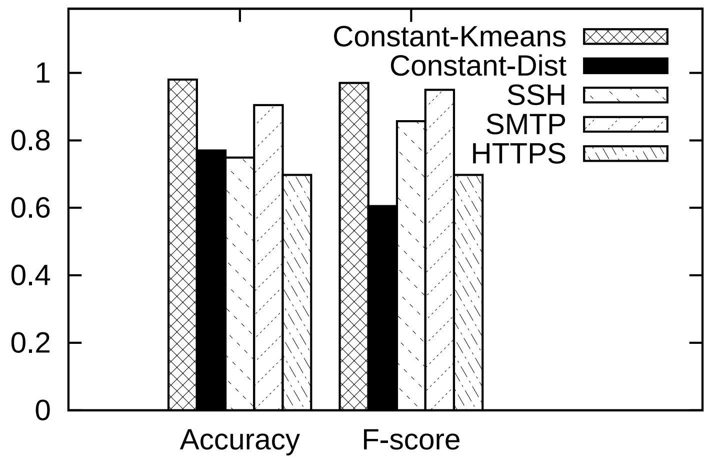


Figure 3.2: Classification attack results for generated cover traffic

traces as described in Section 2.1. For comparison to these realistic protocols models, we also use TrafficMimic to generate constant-rate cover traffic. TrafficMimic sends a full size packet (1448 bytes) every 500ms. The slave TrafficMimic client echoes back the same size packet upon receiving a packet from the master. Ten such exchanges occur per connection before a new connection starts. This constant-rate model is a trade-off between interactivity, bandwidth, and overhead. In future work we plan to also evaluate randomized traffic models. We take the generated cover traffic and apply the classifier using the training data from a different time period of traffic (Jan CAIDA and April 20).

When using TrafficMimic, we were able to fool our classifier approximately 73% of the time for the three realistic protocols we generated (Figure 3.2). Thus, on average, our generated protocols are properly detected as real protocols at 91% of the rate that real protocols are. This is well within the error range for our classifier in the general attack scenario described above and is unlikely to lead to reliable defense detection.

We also extend our classification attack to support a simple anomaly detection algorithm as follows. During training, we find the maximum distance of any training point in each class from the centroid of the class (e.g., m_k where k enumerates the classes in the training set). During the application phase, we consider any test example whose distance, d_i, to its nearest neighbor is greater than the maximum distance observed during training to be an anomaly. We use the threshold t to control the degree to which test examples are labeled unknown. Thus a test example is considered an unknown protocol if: $d_i > t * m_k$.

Illustrating the vulnerability to defense detection of constant-rate schemes, we found that we were 77.1% accurate in detecting constant-rate traffic as anomalous (using the max distance detector with t=1.1). We found that the distance threshold algorithm was not able to differentiate SMTP and constant-rate traffic as well as it was for HTTPS, SSH, and constant-rate. To solve this problem, we introduce another simple unsupervised anomaly detector to be used for port 25 only. It uses the K-means algorithm to cluster the data. It then uses a threshold algorithm to determine which clusters have the lowest mean inter-cluster distance. Any clusters that have greater than the threshold inter-cluster distance are considered anomalous. We found that this technique was 95%+ accurate at differentiating SMTP and constant-rate traffic.

3.2.2 TrafficMimic-Tailored Attack

We next test TrafficMimic's ability resist a defense detection attack *specifically designed* to identify TrafficMimic generated traffic. To conduct the attack, the attacker constructs a binary classifier that is trained to differentiate real traffic of a given protocol from TrafficMimic generated traffic mimicking the same protocol. Thus, the attacker has access to TrafficMimic and may use it generate training data for the attack.

To create training data, the attacker first samples real traffic of protocol X from a generic trace (in our case we use HTTPS from the February CAIDA trace data). Then the attacker takes the same generic trace and trains TrafficMimic behavior models for protocol X. The attacker then generates cover traffic using the X model over a monitored network link (IL-CA). The attacker combines these two sets of training data in a manner that preserves an equal base-rate (i.e., half real and half cover). The attacker can now use this training data to differentiate fake and real traffic of protocol X using the same protocol classification attack described in Section 3.2.1.

To evaluate the effectiveness of this attack we use TrafficMimic to generate a second set of traffic, which is then tested against the trained classifier. We use a distinct (but generic) network trace to create behavior models for the defender (i.e., LANL traces). Thus we do not allow the attacker to train on exactly the same trace as the defender. We then test it over the MN-UK link because in the worst, but most realistic, scenario the attacker does not have access to the same network link as the defender. We found the attack to be no more effective than random guessing (49.4% accuracy).

3.3 Performance and Overhead Results

We next investigate the performance of TrafficMimic as it compares to a constant-rate defense. We use two simple mechanisms to generate real traffic: an SSH-based interactivity model and bulk transfer. While these traffic patterns do not reflect real network usage, they provide insight into the performance and overhead properties of TrafficMimic. We believe that the interdependencies between using a complex traffic generator like Swing to both load the system with real traffic *and* generate cover traffic to carry it would result in data that were difficult to interpret. We repeat each of the experiments below at least 10 times.

3.3.1 Interactive Performance

To evaluate interactive traffic in a TrafficMimic tunnel, we use a traffic model inspired by our prior work [52]. Instead of analytically modeling the distributions of packet sizes and inter-packet times, we extract these features directly from SSH traces from the January CAIDA data set. Since SSH includes both interactive uses (e.g., typing in a terminal) and other non-interactive features (e.g., file transfer or X forwarding), we focus on extracting *only* the features resulting from a user typing into an SSH console.

In a standard SSH connection, each user key press generates an encrypted packet to the server. The server immediately responds with another encrypted packet that contains the character to be displayed on the user's terminal. Since these key-press/response packets require only a few bytes to transmit, the packets sent over the wire are the

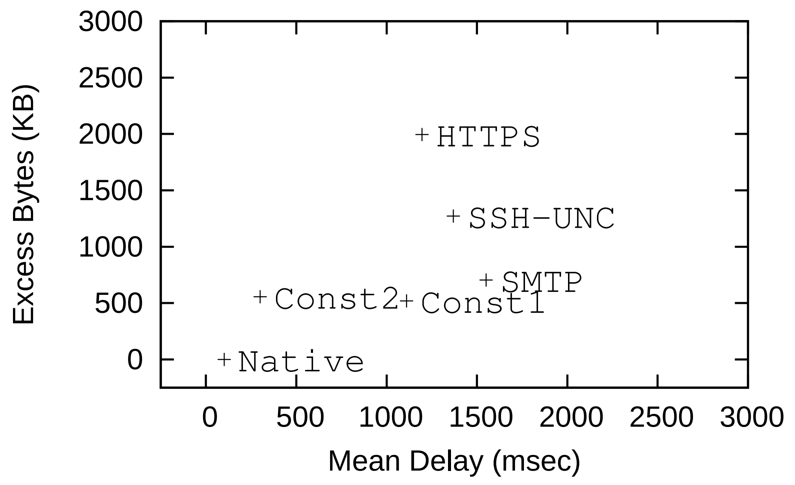


Figure 3.3: Overhead of generated protocols carrying interactive traffic on MN-UK link

minimum size for the encryption and MAC algorithm chosen by SSH (28-52 bytes depending on algorithm). We searched for exchanges of packets in SSH connections that used one of the possible minimum packet sizes for both the key-press and key-response packets. We compute the latency between these key exchanges to find the user think time between key presses.

We created a simple echo utility that randomly samples the interactive SSH data we extracted and emits appropriately sized packets with the timing from the trace. We then connect this utility to TrafficMimic with various generated cover traffic models. In addition to using the constant rate model described above (Const1), we also created a second constant-rate model that had a faster inter-packet time (Const2). We ran the echo utility without TrafficMimic to find the mean latency of the network link (native). We show the results of this experiment using the MN-UK link in Figure 3.3. We found that we were able to achieve a mean delay comparable to the constant-rate schemes with only a moderate increase in excess bytes. Surprisingly, protocols that inherently do not seem to favor sustained interactivity such as SMTP and HTTPS both offer reasonable delay compared to the constant-rate models.

While it was easy to improve the performance of the constant-rate model without sacrificing overhead (Const2), realistic traffic patterns would make this tuning more

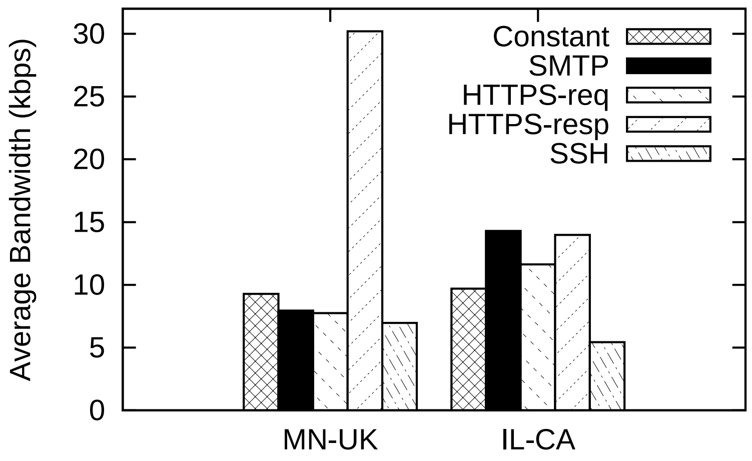


Figure 3.4: Bandwidth performance of generated protocols carrying bulk transfer traffic

difficult. For example, long periods of network silence would generate much higher overhead with the faster acting constant-rate stream. In an empirical test of real user Web browsing with Const1 and Const2 showed up to a 3.4x increase in overhead without appreciable performance improvement. Furthermore, cover traffic tunnels, either realistic or constant, still significantly impact mean interactive latency.

3.3.2 Bulk Transfer

To test the bandwidth of cover traffic tunneling we use Iperf [53] to generate a simple one-way bulk transfer of a 100KB file. Since HTTPS is generally more asymmetric in the number of bytes transferred from vs. to the server, we also perform the bulk transfer from the slave to the master TrafficMimic node (HTTPS-resp). For the other protocols and constant-rate models, we perform the bulk transfer from the master to the slave node. Figure 3.4 shows the bandwidth in kbps for the entire bulk transfer on the wide area test links. Figure 3.5 shows the ratio of the transmitted bytes to the original 100KB test file size. This figure includes *both directions* of the cover traffic even though the bulk transfer is only one-way.

The results show that we are able to achieve comparable or better bandwidth using realistic protocol models as compared to the constant-rate model. The performance of

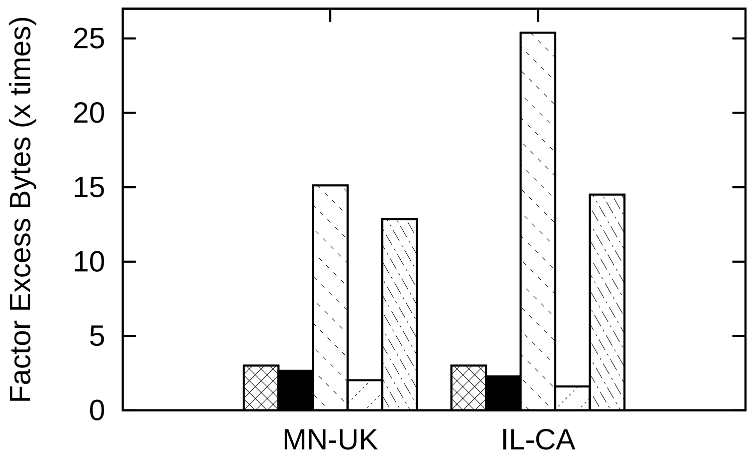


Figure 3.5: Overhead of generated protocols carrying bulk transfer traffic

HTTPS is especially good because the return path in the HTTP(S) protocol inherently favors bulk transfer of files. The SSH traffic model showed much higher overhead. This is primarily due to the asymmetry of SSH connections in how much flows from server to client. Overall, realistic protocol models provide several options for performing bulk transfer efficiently while not succumbing to the security problems associated with defense detection. In Chapter 5, we investigate how to optimize the generation of realistic cover traffic depending on the properties of the real flow and the needs of the user.

3.3.3 Web Browsing

To illustrate some of the performance properties of TrafficMimic we use it with various traffic models to browse 5 common Internet sites. We retrieve the pages using a tunnel on the MN-UK link. We use an OpenSSH tunnel to approximate the native performance of proxying through these hosts. We show in Figure 3.6 the slowdown of each protocol behavior model compared to the native proxy. The HTTPS traffic model is better suited to carrying Web traffic than the generic constant-rate model. We also found the SSH client model to be effective at transmitting Web traffic. We do not show the results for SMTP because its performance carrying Web traffic was very poor (80-

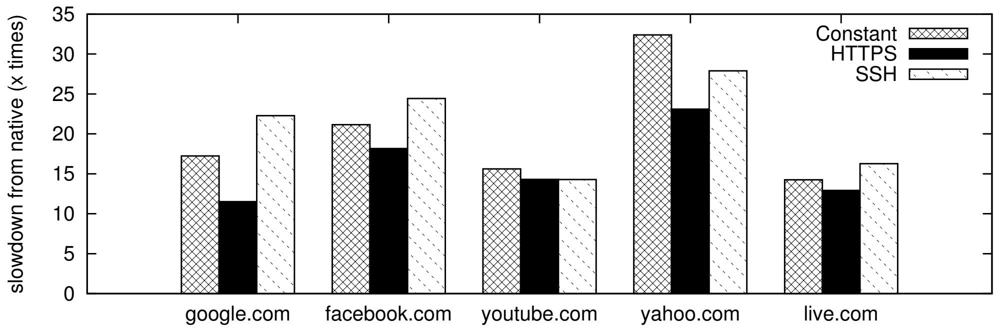


Figure 3.6: Web page load times using cover traffic models

150x slowdown). We believe this is due to the long sleep times that can occur in SMTP that are mismatched with interleaved Web requests.

3.4 Conclusions

To evaluate TrafficMimic, we developed a new traffic classification algorithm based, in part, on previous work in the area. We showed that the traffic models used in TrafficMimic result in detection rates that are similar to those of real traffic and thus provide a good countermeasure for defense detection. We also evaluated the performance of TrafficMimic for both interactive and bulk-transfer protocols and compared it with constant-rate cover traffic models. Overall, we found that TrafficMimic offered reasonable performance; in a later chapter of this work, we investigate how to parameterize traffic generation models to better fit the real traffic being tunneled, and the effect this has on both performance and security.

CHAPTER 4

UNDERSTANDING TRAFFICMIMIC PERFORMANCE

We thus far presented the design and evaluation of independent realistic cover traffic tunneling. We next turn to simulation and analytic modeling to more deeply understand the performance properties of our approach. In this chapter[1], we investigate what effect realistic cover traffic tunneling has on the real traffic being sent. Specifically, what happens when the cover traffic and real traffic differ greatly in rate, and in the fundamental carry size? Under what conditions would the latency of the real traffic be significantly affected by various cover traffic? Under what conditions can we ensure that the throughput of the real traffic is not affected by tunneling?

We developed a simulation model of this type of scenario using SSFNet [18]. The model has detail with respect to protocols running on the hosts, but is simple in its representation of the network. This chapter describes the model, and results analyzing the effects of tunneling. It then develops an analytic model, which assumes tunneling is carried out on a per-session basis. The model captures tunneling of both the request, and the response elements of the traffic in a manner similar to the HTTP protocol. The model characterizes performance as a function of traffic characteristics, and validates the model's predictions against measurements observed in the simulator. Using the validated model, we determine conditions under which real traffic throughput is not impacted by tunneling (although latency will always be impacted), and quantitatively evaluate the impact on real traffic latency (e.g., "slowdown") that tunneling imposes. We develop a Markov chain model of this system, use it to determine conditions under

[1]Material from this chapter has, in part, appeared before in *SIMULATION: Transactions of The Society for Modeling and Simulation International* [54]. This chapter is also based, in part, on materials in *Proceedings of the 2008 Conference on Principles of Advanced and Distributed Simulation* [55]. The authors retain partial copyright and approve the use of the previously published materials in this dissertation. David M. Nicol contributed to the data, analysis, figures, and tables in Sections 4.2 and 4.3.

which real traffic throughput is not affected, and use it to quantitatively explore how real traffic latency is affect by tunneling.

4.1 Simulation

To develop a tractable simulation model for TrafficMimic, we model all cover and real traffic as request/response pairs. Each request/response pair is a single *flow*. This model captures the behavior of many protocols and provides a consistent means to specify both model and real traffic. The size of requests and responses can be read from a trace file, or be generated randomly from given probability distributions.

In the simulator, we consider there to be two peer hosts within the system which generate cover traffic to mask the real traffic they are transmitting to each other. We assume that the attacker only has access to the network on which the real traffic is obscured by the cover traffic. If this system is used as a proxy service, we do not protect the real traffic from analysis before it is received and tunneled by one of the proxy end-points. For simplicity in this simulation we assume that both the cover traffic and real traffic originate on the same host.

We use similar terminology for the components of the simulation model as we do for the implementation of TrafficMimic as described in Section 2.2. The *tm-client* is the host that originates the cover connection and issues the request. The cover traffic *tm-server* sends a response to the client after receiving its cover request. As we are interested in impacts on real traffic that are much larger in magnitude than network latencies, we need not (and do not) model the communication network between host and client with any significant detail. Furthermore, we will see that the traffic itself has high variance, which implies that the behavior already has significant effects we would expect to see using a more detailed network model.

The core simulation loop continually plays back the cover traffic flows over simulated TCP connections. A cover traffic flow has the following steps:

1. tm-client issues a cover request to the tm-server

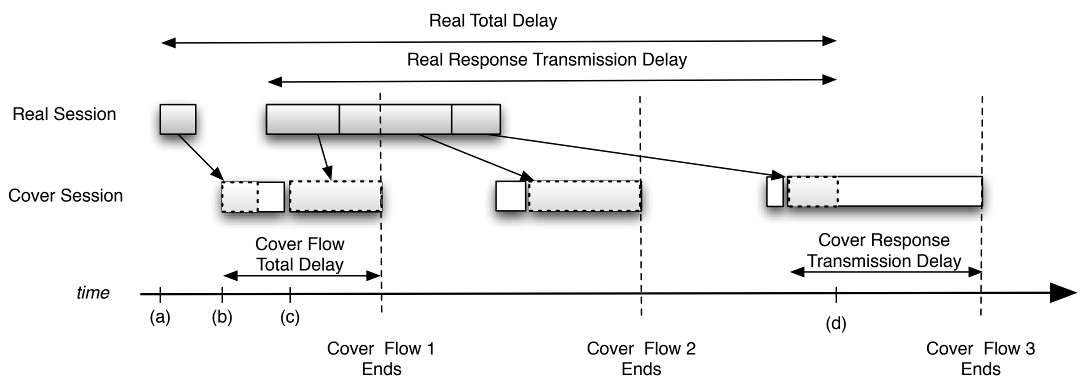


Figure 4.1: Example of interaction between cover and real flows

2. tm-server sends a cover response back to the client

3. tm-client pauses for randomly distributed inter-cover flow time and then repeats.

The sizes of the request and response may vary from session to session. We obtain baseline performance of the real traffic by using these cover traffic flows. We time-stamp each request and response at creation to calculate its delay. Later we can compare the baseline delay against the performance of the same traffic embedded in a tunnel.

While the simulator is generating cover flows, it also sends available real traffic over the tunnel. Similarly to TrafficMimic, either the tm-client or tm-server may be the originator for the real traffic. The real traffic client host is called the *master* and the server is called the *slave*. The master generates a real request (we will call *A*) for a certain number of bytes to be sent after a random amount of time into the simulation. Since the simulator is already sending cover traffic from the start of the simulation, this will ensure that *A* will begin either during or between existing cover flows. We mark *A* with its creation time-stamp, so that when it is delivered we can compute the total delay. The simulator queues *A* until the cover flow next transmits data. *A* can be embedded in either a request or response of the cover traffic. The system uses the size of the cover traffic transmission to consume the correct number of bytes from *A*. This process continues until *A* is fully sent (potentially across multiple distinct cover flows). There may be additional cover flow padding to be transmitted if the remaining data in *A* is smaller than the final cover message that carries it. Once the slave receives *A*, it generates and sends a response, *B* in a similar fashion. At the end of the simulation, the master and slave each report the throughput and delay of *A* and *B*.

We illustrate an example of this process in Figure 4.1. We represent the cover session as a series of flows with a real session above them in the diagram. The cover session is inactive when the real request arrives, so it waits for the cover session to begin again. The real response is split across three cover flows and takes considerably longer to transmit than the real request. The real flow completes before the last cover flow that carries it because the real data to be transmitted is smaller than the final cover response that carries it.

Figure 4.1 also introduces several delay metrics we use to describe the performance of cover and real flows. We use the term *transmission delay* to describe the time it takes to transmit a request or response of *X* bytes. This delay starts when the request or response is created and ends when it is fully received by the other end-point of the connection. Cover transmission delay represents only the time required to transmit *X* bytes across the network using TCP. Real transmission delay may span several cover flows, therefore may also include TCP connection setup/tear-down and inter-cover session times. We call the time required to complete both the request and response of a single flow the *total delay*. Again, real total delay may span several cover flows and may include time that is not directly due to transmitting data across the network.

4.1.1 Implementation

We created the system model described above using SSFNet [18]. The model contains two protocol hosts which use blocking sockets running TCP. We adapted the standard blocking socket TCP server and client to play back cover flows automatically, and also play back generated model traffic if so specified in the model configuration file. We specify distributions and moments or empirical CDFs for request and response sizes which the server and client will use. The inter-cover flow time is drawn from an exponential distribution with mean 200ms.

The entity representing the tm-client communicates certain information to the entity representing the tm-server to govern the generation of the simulated message sizes. The tm-client entity specifies the sending time, the size of the request, the size of the response, the type of the message, and the number of bytes from the real traffic to be tunneled. Given that our TrafficMimic implementation imposes minimal overhead from its forwarding and fragmentation protocol, our simulator models an ideal tunnel implementation where space consumed by this per-message overhead does not reduce the amount of space available in a cover message for real traffic. Data sizes of the request and the response are sampled from empirical probability distributions based on real observations.

Both tm-client and tm-server use timers to implement the start of the real traffic. When fired, the timer's callback code generates either a request or a response, as appropriate. Since each simulation will only show the effect of a single real flow, the model needs only to record the aggregate amount of requested traffic to send, and the aggregate amount of traffic that has been received. This captures the queued real message's time waiting for enough tunnel traffic to be communicated.

Each message also has a state code, which specifies the connection state (new, empty, and continue). The empty state simply tells the server to ignore the message because no real traffic needs to be transmitted. This cover flow is entirely padding. The new state instructs the slave that a new real flow is starting and gives the parameters for its behavior. The continue state encodes the number of real bytes sent in an existing real flow. Both master and slave keep track of the number of bytes remaining to be received on a real flow. When receiving part of a real message, the slave determines whether the message is a request or a response by a flag in the message called initiator. It then decrements the local state of how many bytes remain to finish the message. If the message has been completed, it will print a report on the overall throughput of the system and then generate a response. Minor bookkeeping is needed to support computation of real traffic bandwidth and latency.

4.1.2 Generating Realistic Traffic

In this study, we focus on the behavior of HTTP traffic because it is the dominant protocol used on the Internet today. Furthermore its encrypted equivalent, HTTPS, is the most widely used encrypted protocol. We consider the specific behaviors of HTTP since the TLS tunnel used by HTTPS does not change the observable patterns of the protocol[2]. As stated, the typical pattern for HTTP is a small request followed by a larger response. While this does not capture the full spectrum of what HTTP can do, it provides a structure for the most common use of the protocol.

To determine the size distributions for requests and responses, we collected data from real network traces. We assumed that all traffic on port 80 was HTTP. Since we did not have access to packet payloads, we were unable to validate this assumption. After performing appropriate TCP stream reassembly, we used several heuristics to differentiate requests and response from a live network trace. All bytes sent to port 80 were requests and bytes from port 80 were responses. Since requests and especially responses can be split across many individual IP packets, we coalesce the packets together. All non-zero length payloads sent in one direction without a non-zero length payload sent in the other direction are coalesced together as one. This allows for the TCP stack to transmit acknowledgments without arbitrarily splitting objects. Due to network reordering and congestion, this heuristic may incorrectly classify request and responses. However, we assume this to be rare enough to not skew the response size distributions considerably.

We used the above methods to analyze two hour-long traces collected by the University of North Carolina campus-wide border link in April 2003 [49]. While the data from these two traces are similar, we use the data from one trace to generate cover traffic and the other trace for real traffic. This allows us to keep the cover and real traffic independent since we randomly sample real observations from the traces to generate responses sizes.

To find the distribution of request sizes, we eliminated large outliers in the data. These larger requests were likely HTTP POST commands, which contained a variable

[2]We ignore the behavior of TLS itself in this analysis; specifically, session start-up, key exchange, and session tear-down.

length payload. Even after removing large outliers, we retained over 90% of the data in both traces. We observed that the request sizes were Gaussian and verified this using the Shapiro-Wilk normality test. The mean request size from these traces, approximately 400 bytes, also matched intuition based on the number and length of HTTP headers used by common Web browsers.

We found the distribution for response sizes to be more complicated. Prior work has estimated that responses are a mix of a log-normal distribution and a long-tailed distribution such as Pareto [39]. In the traces we analyzed, we observed significant variation in response sizes, so we chose to use empirical distributions rather than fitting a known distribution. Since some types of HTTP responses only include headers (e.g., for cache control), we focused on the upper quartile of the response data (greater than $\sim$4KB). This ensured that we were capturing actual Web content rather than HTTP headers or XMLHttpRequests generated by AJAX. We also excluded very large responses (greater than 1MB) as they likely represented file downloads rather than Web page browsing. The large files we excluded represented less than 1% of the data.

Intuitively, response sizes vary greatly depending on the nature of the content being browsed. For example, browsing a text-only web page will exhibit significantly different object sizes than viewing a page with high-resolution images or videos. We used K-means clustering to identify three different categories and respective empirical distributions. We call the categories low, medium, and high with respect to the magnitude of their center points. While the distinction between these categories is purely arbitrary, they establish some reasonable classes of response sizes to evaluate disparities between different traffic types. Thus, we can evaluate the performance resulting from different mixtures of these categories for the cover and real traffic. We give descriptions of the clusters from both traces we analyzed in Table 4.1.

4.1.3 Results

Our simulation evaluation uses a simple network that connects two routers with a 50ms delay and 100Mbits/s bandwidth link; each router has one attached host (client, and

Table 4.1: Response size distribution clusters from UNC Web traffic traces

(a) Real traffic–collected April 20, 2003

	Center	Count	Min	Max
low	15206	395972	3779	76440
med	137702	25592	76458	378673
high	619700	2761	378798	1048352

(b) Cover traffic–collected April 29, 2003

	Center	Count	Min	Max
low	15441	437974	4250	100808
med	186254	14112	100849	432750
high	679303	2889	432826	1048463

server); host to router links have 20ms delay and 1.5Mbits/s bandwidth (Figure 4.2). While simple, this network provides a useful baseline to quantify the effects of tunneling on a normal Internet user.

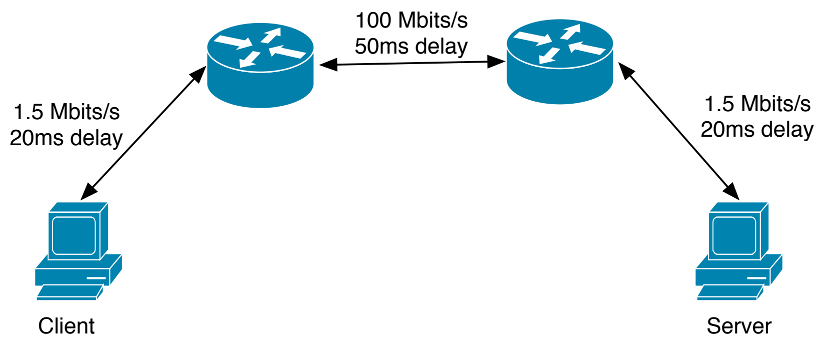


Figure 4.2: Simulated network configuration.

We evaluated the performance of cover sessions with the parameters from the three response size categories on the network configuration specified above to establish a baseline for TCP performance on our network. We show the results for transmission throughput in Table 4.2. This data is computed by averaging the results of many experiments. Each experiment randomly chooses a size sample from the empirical distribution associated with its type, then measures the time needed transmit the data (e.g., the transmission delay). Data size divided by the measured transmission delay gives one throughput sample. Table 4.2 gives the sample means and standard deviations from

Table 4.2: Baseline transmission throughput (bytes/sec)

	request	low	medium	high
mean	4243	21453	99643	148963
std dev	1449	11613	17138	6468

Table 4.3: Total delay (seconds) on a real session tunneled over HTTP cover traffic

Traffic Types	Mean Total Delay	Std Dev
low/low	2.899	1.683
low/medium	3.394	1.767
low/high	6.399	3.621
medium/low	12.666	5.716
medium/medium	4.647	1.948
medium/high	7.327	3.658
high/low	53.053	16.486
high/medium	11.179	3.165
high/high	10.299	3.809

these experiments. Despite the large variation, we based the statistics on so many samples that the width of the 95% confidence interval is very small (less than 1% of the mean).

These values capture the impact that the TCP protocol has on transfer rates—the larger the segment sent, the lower the per-byte cost is. Our analytic model will use these baseline measurements to account for size-dependent throughput.

To evaluate the effects of tunneling, we layered one traffic model on top of another. To denote each tunnel test, we specify a traffic model that runs the real traffic first followed by a slash and then the cover traffic model. For example, we call a test running medium load real traffic over low load cover traffic: medium/low. The metric of interest is total delay of a real flow—the difference between when a real flow's request is ready to be carried, and when the last byte of the real flow's response is delivered.

We performed simulation experiments to assess the effects of tunneling real traffic running HTTP over a tunnel that also uses HTTP. Table 4.3 gives the sample mean and standard deviation of the total delay associated with serving that request and its corresponding response, as a function of the real and cover traffic types. We will use

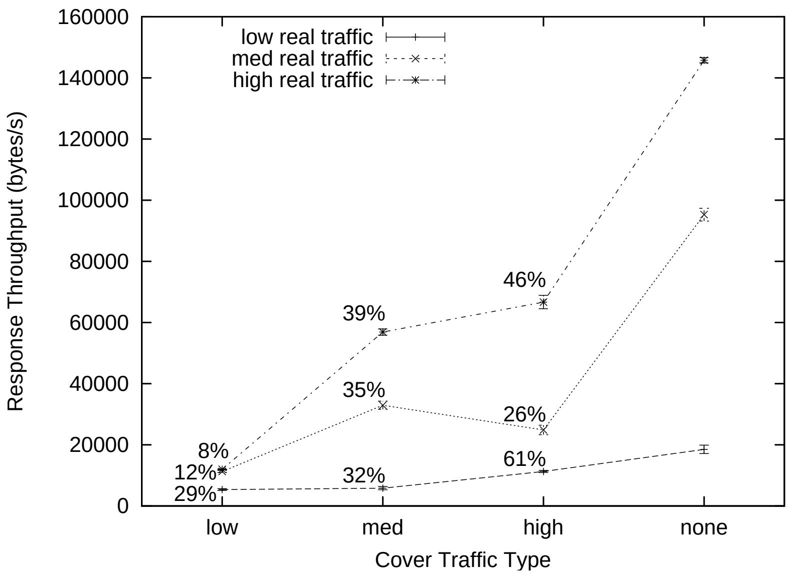


Figure 4.3: "Response Throughput" of HTTP traffic tunneled over HTTP cover traffic

data in this table as the basis for comparison with an analytic model that predicts total delay. Figure 4.3 shows a different metric associated with these same experiments, the mean "response throughput", where each sample measures the total number of bytes delivered *by the response only*, divided by the *total delay* from arrival of the real traffic request to the time when the last byte of the response is delivered. The idea is to focus attention on the essential payload of the transaction, and treat the transmission of the HTTP request as part of the set-up overhead, and not part of the delivered data. Each real traffic type is shown by a series on the graph and the cover traffic type is indicated on the x-axis. The error bars denote the 95% confidence interval around the sample mean. The "none" category is the response throughput of the native traffic type, i.e., when the cover flow is not used. Each data point corresponding to a cover flow is marked with the size of the corresponding throughput response, relative to the native relative throughput.

We found that for low and high load real traffic, response throughput increases monotonically with increasing bandwidth cover models. The sole exception to this trend is the data point where medium load real traffic is carried by heavy load cover traffic. The way to understand why the trend need not be uniform across all traffic types is the realization that with increasing cover bandwidth there comes both the performance enhancing aspect of greater transmission efficiencies due to longer response transmissions, but at the same time longer average wait times by the real traffic waiting for the current cover session to end before its own request is served. The data otherwise confirms intuition; that low bandwidth cover traffic severely impacts performance (heavy load real traffic gets only 8% of native response throughput, and even low load real traffic gets only 29%); on the other hand, with heavy load cover traffic the performance is (very approximately) half that of the native traffic for two of the traffic types, and a quarter for the other. We improved the performance results compared to our previous results [55] by delivering real data as soon as it is received rather than waiting for the cover session, which carries the data to complete. We later apply this enhancement to the TrafficMimic implementation in Chapter 5.

Overall, we observe substantial decreases in performance due to tunneling, with some sensitivity to the cover traffic type. This indicates a need to carefully choose the cover traffic model to fit the needs of the real traffic.

4.2 Analytic Model

A simple analytic model allows us to explore the inter-relationships between model parameters and how the characteristics of the cover flow and real flow interact to affect the performance of the real flow when tunneled. We develop the model, validate it, and use it to derive bounds on the performance of tunneled traffic, and to describe constraints on the cover flow that are necessary to ensure that the real traffic when tunneled is not delayed beyond its natural arrival rate.

4.2.1 Model Details

We suppose that the data sizes of both cover flow and real flow sessions are measured in a common unit, bytes, and that with respect to these units, the size of a session is a random number of units. We assume that the distribution of an HTTP request size is the same for both cover and real flows, and denote that variable as G_{req}. The size of an HTTP response for the cover flow is $G^{(c)}_{rsp}$, and the size of an HTTP response for the real flow is $G^{(r)}_{rsp}$. The time required to transmit a byte of a HTTP request is denoted τ_{req}, while the time required to transmit a response byte of the cover flow is denoted τ_c, and the corresponding time for a real flow's response byte sent without tunneling is τ_r. These constants incorporate all the effects of the network on packet transmission, e.g., TCP congestion control, packet loss, and link bandwidths. We distinguish between τ_{req} and τ_c because the length of the messages sent may be different, and so the average cost-per-byte will be different due to TCP congestion control.

Tunneled real flow performance depends in part on the delays between cover sessions, which we assume are random, independent, and exponentially distributed, denoted I. The mean cover flow inter-session delay is $E[I] = \mu_c$.

We can view the process of a cover flow as an alternating "on-off" renewal process [56], where the off time is exponential with mean μ_c, and the on time has the distribution of $T_c = (\tau_{req}G_{req} + \tau_c G^{(c)}_{rsp})$. We introduce the notation of T_c for simplicity of expression needed later, and note that

$$E[T_c] = \tau_{req}E[G_{req}] + \tau_c E[G^{(c)}_{rsp}]$$

and

$$E[T_c^2] \quad = \quad \tau_{req}^2 E[G_{req}^2] + 2\tau_{req}\tau_c E[G_{req}]E[G^{(c)}_{rsp}] + \tau_c^2 E[(G^{(c)}_{rsp})^2].$$

The total delay associated with a real session has three components. The first is the waiting time between when the real session arrives to be served, and when the next cover session starts by carrying some (or all) of the real session's HTTP request in a cover HTTP request, we call this random time R_{wait}. On the time-line of Figure 4.1 R_{wait} is the interval between instant (a) and instant (b). The second component starts where the first left off, and ends at the instant where the first cover response begins to carry the real response; we call this R_{req}. On the time-line of Figure 4.1 this is equal to the interval from instant (b) to (c). The final component picks up there, and ends when the last byte of the real response is received; we call this delay R_{rsp}—on the example the gap from (c) to (d). The mean total delay for the real session is $E[R_{wait}] + E[R_{req}] + E[R_{rsp}]$. We now consider each of these terms.

If the real session arrives while the cover session is busy, then it waits for the cover session to end, and then waits for a full inter-session delay to pass. If the real session arrives while the cover session is idle, then it waits only for the remaining inter-session delay. But since that delay is exponentially distributed, the residual delay is as well. In either case R_{wait} contains an exponentially distributed intersession delay. Under our assumptions, the cover session has run to "equilibrium" by the time the real traffic request arrives, then by renewal theory [56] the probability of the cover session being busy when the real session arrives is the ratio of its mean on time to the sum of the mean on and mean off times. If the request arrives when the cover session is busy, the mean time until the cover session ends is the "mean excess life" [56] of the random variable T_c (defined above), known to be $E[T_c^2]/(2E[T_c])$, which, applied to our model, enables us to write

$$E[R_{wait}] = \mu_c + \left(\frac{E[T_c]}{E[T_c] + \mu_c}\right)\left(\frac{E[T_c^2]}{2E[T_c]}\right). \tag{4.1}$$

The expression of $E[R_{req}]$ is conceptually more complex. Depending on the size of the HTTP request presented by the real flow, it may require multiple *full* cover sessions to carry the real HTTP request embedded in the cover HTTP requests. Denote this random variable by C_{req}; in Figure 4.1 it happens that $C_{req} = 0$ because the real request is entirely carried in the first cover session request. The distribution of C_{req} is defined in terms of the distributions of the sizes of the real and cover HTTP request blocks. For simplicity of expression (and based on empirical observation) we assume that the real and cover request block sizes are independent and identically distributed. If we then define a sequence of i.i.d. samples of G_{req} as $G_{req,0}, G_{req,1}, G_{req,2}, \ldots$, then the distribution of C_{req} is seen to be given by

$$\begin{aligned}
\Pr\{C_{req} < n\} &= \Pr\{\sum_{i=1}^{n} G_{req,i} < G_{req,0}\} \\
&= \Pr\{0 < G_{req,0} - \sum_{i=1}^{n} G_{req,i}\} \qquad (4.2)
\end{aligned}$$

While this expression is slightly daunting in the general case, we observe that under the assumption (verified empirically) that G_{req} has a Gaussian distribution, then the sum $G_{req,0} - \sum_{i=1}^{n} G_{req,i}$ is Gaussian, with mean $(1-n)E[G_{req}]$ and variance $nVar(G_{req})$. Therefore, given empirically determined estimates of $E[G_{req}]$ and $Var(G_{req})$ we can easily compute the distribution of C_{req}.

We can express the mean value of R_{req} as

$$E[R_{req}] = E[\sum_{i=1}^{C_{req}} (I + T_c)] + \tau_{req} E[G_{req}],$$

recalling that I is the random inter-session delay for the cover traffic. This expression recognizes that R_{req} is comprised of C_{req} full cover sessions and inter-session delays (because the real request block is not yet fully carried), followed by one final cover request (that carries the last piece of the real request). Now C_{req} is technically a "stopping time" [56], so that Wald's Lemma allows us to simplify the above as

$$E[R_{req}] = E[C_{req}](\mu_c + E[T_c]) + \tau_{req} E[G_{req}]. \qquad (4.3)$$

Finally, the derivation of $E[R_{rsp}]$ follows the logic of $E[R_{req}]$: a random number of full cover session cycles are needed to carry real response bytes, but on the last session the final segment of response bytes are carried. We denote the random number of full cover sessions needed by C_{rsp}; in the example of Figure 4.1 we have $C_{rsp} = 2$. Following the same logic as before observe that the distribution of C_{rsp} is given by

$$\Pr\{C_{rsp} < n\} = \Pr\{0 < G^{(r)}_{rsp} - \sum_{i=1}^{n} G^{(c)}_{rsp,i}\}$$

where $\{G^{(c)}_{rsp,i}\}$, $i = 1,2,3,\ldots$ is an i.i.d. sequence of instances of cover response block sizes $G^{(c)}_{rsp}$. We can segregate the time spent carrying real response traffic into the time spent in actual transmission, and time spent in the other phases of the cover session that are not directly carrying real response traffic. The mean time spent actually transmitting real response bytes is just $\tau_c E[G^{(r)}_{rsp}]$. Here it is important to recognize that the per-byte transmission cost is that of the cover flow, **not** that of the native real flow. The extra time in the cover session is comprised of a sum of C_{rsp} instances of cover inter-session and cover request transmission times. C_{rsp} is a stopping time with computable mean, just as C_{req} was, which brings us to the expression

$$E[R_{rsp}] = E[C_{rsp}]\left(\mu_c + \tau_{req}E[G_{req}]\right) + \tau_c E[G^{(r)}_{rsp}]. \tag{4.4}$$

Using equations 4.1, 4.3, and 4.4 we have an expression for the mean delay a real session has under the tunneling scheme:

$$E[D_{real}] = E[R_{wait}] + E[R_{req}] + E[R_{rsp}]. \tag{4.5}$$

While the expression is exact, it is important to know that in practical application some approximations will be needed. The exact distributions of C_{rsp} and C_{req} depend on the exact distributions of G_{req}, $G^{(r)}_{rsp}$, and $G^{(c)}_{rsp}$. We have good reason to model G_{req} as having a Gaussian distribution. We know that $G^{(c)}_{rsp}$ and $G^{(r)}_{rsp}$ emphatically do not. When known, the distributions of C_{rsp} and C_{req} might be computed to whatever accuracy that is needed. Still, in the validation study we consider next, we use the empirically observed *mean and variance* of those sizes, and *assume* that the distributions are Gaussian. This greatly simplifies the computation of the distributions of C_{rsp} and C_{req}, and hence computation of their mean values.

4.2.2 Validation

Before we use the analytic model to explore system behavior, we consider first how well it describes the simulation data we have collected already.

Section 4.1.2 describes how we gathered request and response data sizes from real traffic, and from these categorized each HTTP response as being in a “low”, “medium”, or “high” load category. We observed that request data sizes were well described by Gaussian distributions, but that the response data sizes were not. Table 4.1 gives the mean and standard deviation of the measured response data sizes, and Table 4.3 gives the simulation results estimating a real session’s mean delay, as a function of the cover and real traffic type. It is important to note that the simulation sampled the response sizes *from the empirical distribution based on a real network trace*. Specifically, every time the simulation sampled a response size, it chose one of the empirically observed session response sizes uniformly at random.

Equation 4.5 expresses our analytic expression for mean delay, in terms given by equations 4.1, 4.3, and 4.4. The latter two depend on values of $E[C_{req}]$ and $E[C_{rsp}]$, which in turn depend on the distribution of the real and cover response sizes. To simplify these calculations we will *assume* that the response size distributions are Gaussian, even though the empirical distributions fail standard statistical tests for the Gaussian. Under this approximation we can compute the mean delays predicted by our model, and compute the *relative* error: $(d_{pred} - d_{obs})/d_{obs}$, where d_{pred} is the delay predicted by the model and d_{obs} is the mean delay observed by the simulation. Table 4.4 gives the relative error as a function of the real and cover traffic types, where, as before, we characterize an experiment in terms of the real traffic type, followed by ’/’ and then the cover traffic type. It also gives a standard measure of the quality of a simulation based estimate, also called relative error, equal to the width of the confidence interval divided by the sample mean.

Despite the approximations of $E[C_{req}]$ and $E[C_{rsp}]$, the analytic model predictions match very well with the simulation measurements. 7% absolute error is the largest magnitude observed, and is only slightly larger than the simulation’s own relative error. We believe the “high” load real traffic experiments have the best accuracy because the delay time is dominated by transmission of real response data (in both predicted and measured forms), which our model very accurately captures, and which leads to much smaller variation in the simulation. When the real traffic load is “low” there is a much

Table 4.4: Relative error of analytic model predictions of mean real session delay, and simulation-based estimates

	Relative Error	
Traffic Types	**Model**	**Simulation**
low/low	0.053	0.051
low/medium	-0.067	0.045
low/high	-0.068	0.049
medium/low	0.041	0.039
medium/medium	0.001	0.036
medium/high	-0.073	0.043
high/low	0.003	0.027
high/medium	0.008	0.025
high/high	0.007	0.032

larger contribution to delay in the R_{wait} and R_{req} components, and there is a larger variation in the simulated values as well.

Having now strong confidence in the predictive power of our analytic model, we now use it to better understand how characteristics of cover and real flows may affect performance of the tunneled flow.

4.2.3 Lower Bound on Slowdown

Our first consideration is of the impact tunneling has on real traffic delay as a function of the traffic characteristics. Under certain conditions on the cover traffic we can derive a lower bound on *slowdown*—the degree by which the real traffic is delayed more under tunneling than natively. Towards this end we identify a condition under which $E[D_{real}]$ increases as μ_c increases.

Since $E[D_{real}] = E[R_{wait}] + E[R_{req}] + E[R_{rsp}]$ and each of these is a function of μ_c, consider the derivative $dE[D_{real}]/d\mu_x$. Inspection of equations 4.3 and 4.4 show clearly that $dE[R_{req}]/d\mu_c > 0$ and $dE[R_{rsp}]/d\mu_c > 0$, so we ask under what conditions are we assured that $dE[R_{wait}]/d\mu_c > 0$? Observe from equation 4.1 that

$$\frac{dE[R_{wait}]}{d\mu_c} = 1 - E[T_c^2](E[T_c] + \mu_c)^{-2}/2.$$

A sufficient condition for this derivative to be non-negative is that

$$2(E[T_c]+\mu_c)^2 \geq E[T_c^2].$$

which will certainly be satisfied if after we drop the μ_c term we still have

$$2(E[T_c])^2 \geq E[T_c^2].$$

Now $E[T_c^2] = var(T_c) + (E[T_c])^2$, so the condition above is satisfied if $E[T_c]^2 \geq var(T_c)$. This is essentially a bound on the variation of T_c that is met by a large class of probability distributions, loosely, ones whose variation is not larger than an exponential's (where the square of the mean is identically the variance). This observation leads us to the first result

Lemma 1 *If $E[T_c]^2 \geq var(T_c)$, then $E[D_{real}]$ is a monotone increasing function of μ_c, so that $E[D_{real}]$ is minimized when $\mu_c = 0$.*

One way of thinking about the impact that tunneling has on performance is to consider the ratio of the mean delay of a real traffic session under tunneling to the mean delay run natively. This is a measure of the relative increase in delay due to tunneling. Lemma 1 tells us the numerator is minimized (and so the slowdown is minimized) when $\mu_c = 0$. We reduce the numerator even further assuming $E[C_{req}] = E[C_{rsp}] = 0$. Under these minimizing assumptions the slowdown is at least

$$\frac{E[T_c^2]/(2E[T_c]) + \tau_{req}E[G_{req}] + \tau_c E[G_{rsp}^{(r)}]}{\tau_{req}E[G_{req}] + \tau_r E[G_{rsp}^{(r)}]}.$$

The delay $\tau_{req}E[G_{req}]$ which appears in both the numerator and denominator is, in large flows of interest, small relative to $\tau_c E[G_{rsp}^{(r)}]$ and $\tau_r E[G_{rsp}^{(r)}]$. Another step towards simplifying this ratio is to recognize that since T_c is the sum of $\tau_c G_{rsp}^{(c)}$ with a non-negative random variable, the mean excess of $\tau_c G_{rsp}^{(c)}$ is necessarily smaller than the mean excess of T_c. Thus, making that substitution reduces the ratio further. Under these substitutions then a lower bound on slowdown is

$$\begin{aligned} & \frac{\tau_c^2 E[(G_{rsp}^{(c)})^2]/(2\tau_c E[G_{rsp}^{(c)}]) + \tau_c E[G_{rsp}^{(r)}]}{\tau_r E[G_{rsp}^{(r)}]} \\ = & \left(\frac{\tau_c}{\tau_r}\right)\left(\frac{E[(G_{rsp}^{(c)})^2]}{E[G_{rsp}^{(c)}]E[G_{rsp}^{(r)}]} + 1\right). \end{aligned} \tag{4.6}$$

This leads us to our result on a lower bound on slowdown.

Theorem 1 *Suppose that $E[T_c]^2 \geq var(T_c)$, and that $\tau_{req}E[G_{req}]$ is negligible relative to $\tau_c E[G_{rsp}^{(r)}]$ and $\tau_r E[G_{rsp}^{(r)}]$. Then the ratio of the mean delay of a real session under tunneling to the native mean delay is at least*

$$\left(\frac{\tau_c}{\tau_r}\right)\left(\frac{var(G_{rsp}^{(c)})}{E[G_{rsp}^{(c)}]E[G_{rsp}^{(r)}]} + 1\right) + \frac{\tau_c E[G_{rsp}^{(c)}]}{\tau_r E[G_{rsp}^{(r)}]}.$$

Proof: We use the identity $E[(G_{rsp}^{(c)})^2] = var(G_{rsp}^{(c)}) + E[G_{rsp}^{(c)}]^2$ and equation 4.6 to derive the result. □

The final expression for slowdown exposes a key relationship that impacts it. The ratio $\tau_c E[G_{rsp}^{(c)}]/\tau_r E[G_{rsp}^{(r)}]$ compares the time it takes to transmit a cover session response with the time it takes to natively transmit a real session response. When the cover traffic and real traffic types are identical this ratio is one, $\tau_c = \tau_r$, and so the slowdown is at least two. When the cover session response is significantly larger than the real session response then we expect $\tau_c < \tau_r$, and this slowdown bound is at least the degree to which it takes longer to transmit a cover session response than it does a real session response (natively). When the cover session response is significantly smaller than the real session response then we expect $\tau_c > \tau_r$, and slowdown is at least τ_c/τ_r.

When considering the implications of these equations, it is important to remember that they are predicated on the assumption that $\tau_{req}G_{req}$ is comparatively negligible. For example, in Figure 4.3 we have $\tau_{req}G_{req}$ being approximately 50% percent of $\tau_r G_{rsp}^{(r)}$, so it cannot be viewed as negligible. Still, these simple bounds expressions help to explain the data points associated with the real session under heavier load.

4.3 Markov Chain Model

The dynamics of additional delay suffered by a real session are complex. The formulation shown already is able to express the mean delay a real session suffers, but expresses *only* that mean. For any deeper understanding of what is going on we need a different model. By making some simplifying assumptions we are able to describe dynamics using the theory of Markov chains. Towards this end we ignore the HTTP request phase for both real and cover sessions, and so model the "on" phase as that carrying the HTTP response, with the "off" phase being the (exponential) inter-session delay with mean μ_c. Unlike the earlier analysis, here we consider the real traffic to itself be an on-off process, with exponential off-time having mean μ_r.

We assume the number of bytes sent in a cover session "on" phase is geometrically distributed, so that the time spent in the on phase is a constant times a geometric random variable. We assume that μ_c and μ_r are expressed in units where that constant is 1. We suppose that the mean number of bytes transmitted during a cover session on period has mean $1/p_c$, and that the mean number of bytes required by a real session has mean $1/p_r$.

Under these assumptions we can model and analyze system dynamics formally using the theory of Markov chains. The state of the cover process is always off (0) or on (1). The state of the real process is either off (0), waiting for a cover session to carry more real traffic (1), or being carried by the cover session (2). There are six interesting system states denoted by a tuple of these two state descriptions. On entry to any one of them, the memoryless properties of the exponential and geometric distributions make the future behavior of the system entirely dependent on the state entered. The time within a state need not be exponential though, so the continuous time process is not a standard continuous time Markov chain. However, the system we describe has an embedded discrete-time Markov chain from which we will be able to derive the steady-state state occupancy probabilities of the continuous time stochastic process.

We now describe each state, its outbound transition probabilities, and the mean time the system stays in that state (which is needed for the steady-state probabilities). A

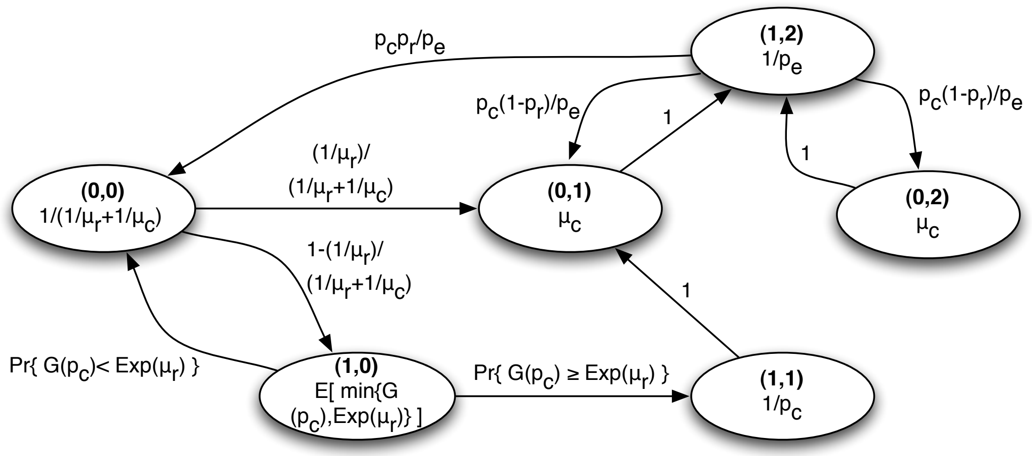


Figure 4.4: Discrete-time Markov chain for session model

diagram of this chain is also shown in Figure 4.4. We denote states as ovals with transition probabilities labeling the arrows. We give the mean waiting time inside each state oval along with its tuple state label **(x,y)**.

(0,0) : Both processes are off, which means that the state changes when the first of the waiting times completes. Since the waiting times are exponential, the probability of transitioning to $(0,1)$ (i.e., the real session arrives first) is $(1/\mu_r)/(1/\mu_r + 1/\mu_c)$. The probability of transitioning to $(1,0)$ is the complement of this—no other transitions from this state are possible. The mean holding time in this state is the mean of the minimum of the two constituent exponentials, e.g., $1/(1/\mu_r + 1/\mu_c)$.

(0,1) : A real session has arrived and is waiting for the cover process to begin. When it does, the system transitions to state $(1,2)$—that's the only transition possible. The mean time the system spends in this state once entered is μ_c.

(1,0) : The cover process starts at a time when the real process is off. From here one of two things can happen. Either the cover process completes before the real process starts (i.e., a transition to $(0,0)$), or a real session arrives to be carried while the cover session is yet active. The transition probability into $(0,0)$ is therefore the probability that a geometric with success probability p_c is less than an exponential with mean μ_r; the transition probability into $(1,1)$ is its complement. The mean time in this state is the mean of the minimum of independent geometric and exponential random variables. These quantities can be computed numerically.

(1,1) : In this state the cover process is active while the real process is waiting. The only transition from this state is to $(0,1)$. The mean time in the state once entered is $1/p_c$.

(1,2) : This is the only system state where real traffic is carried. The system stays in this state until either the cover session ends, or the real session ends. At any given step the probability that at least one of the two sessions ends given that neither has yet is $p_e = 1-(1-p_c)*(1-p_r)$—one minus the probability that they both end. Transitions are possible into state $(0,2)$ (with probability $p_c(1-p_r)/p_e$), into state $(1,0)$ (with probability $p_r(1-p_c)/p_e$), or state $(0,0)$ (with probability $p_c p_r/p_e$).) The mean time in this state once entered is $1/p_e$.

(0,2) : This state is reached when a cover session ends before the real session it was carrying completes. The only transition from here is back into state $(1,2)$, which occurs when the cover process enters the on state again. The mean time in this state once entered is μ_c.

The values of the transition probability matrix P for this system are described above; it is straightforward to numerically solve vector equation $\pi^* = \pi^* P$ to find the equilibrium state occupancy probabilities (in vector π^*), e.g., using Gaussian elimination. By the theory of generalized Markov processes [56], the equilibrium fraction of time the system is in state S is given by $\pi_S = \pi_S^* m_S / \sum_{\text{states } T} \pi_T^* m_T$, where m_S is the mean time in state S, once entered.

4.3.1 Stability

The session model we have explored has the characteristic of holding back real traffic in one session so long as the traffic in the previous session is still being carried. As such, using tunneling could therefore reduce the rate at which real sessions are processed. We aim to answer the following questions: what are the characteristics of the cover flow that are necessary to "keep up" with a real flow, or alternatively, given a cover flow, what real flows can use it to tunnel without having their inherent session throughput adversely affected?

We approach the problem by thinking of the tunneling as a service, and the real sessions to be tunneled as jobs to be served. We again ignore the HTTP request phase. In this model we allow the "off" time of one cycle to coincide with the processing time of a previous cycle. In other words, the waiting time between successive real session transfers may be masked by the processing time of a prior transfer. Viewed this way gives rise to a queueing system with one server, whose mean service time is $E[D_{real}]$. The inter-arrival time between real sessions is the time between the beginnings of real sessions when they are run natively. Continuing to use $1/p_r$ to denote the mean number of bytes in a real session response, and $1/p_c$ to denote the mean number of bytes in a cover session response, the mean of this inter-arrival time is $\tau_r/p_r + \mu_r$, where μ_r is the mean "down" time between successive delivered real sessions, run natively.

We know from the theory of $G/G/1$ queueing systems [56] that the queue is stable (i.e., has a limiting state occupancy distribution) when the service rate is strictly greater than the mean inter-arrival time. We obtain the service rate from our Markov chain model—the fraction of time the system is in state $(1,2)$, times the cover session throughput: $\pi_{(1,2)}/\tau_c$. Stability is ensured when

$$\pi_{(1,2)}/\tau_c > \frac{1.0}{\tau_r/p_r + \mu_r}.$$

In the subsequent analysis of slowdown we use this bound to consider only model parameter sets where the resulting system is stable.

4.3.2 Slowdown

Vector π from the Markov chain model encodes what we need to compute slowdown. Consider: given that the system has a real session to carry, the rate at which that data is carried is the fraction of time the system is in state $(1,2)$, conditioned on there being real traffic to carry, times $1/\tau_c$:

$$\psi_c = (1/\tau_c)\frac{\pi_{(1,2)}}{(1-\pi_{(0,0)}-\pi_{(1,0)})}.$$

The native rate that real session traffic is carried, given that real traffic is available, is $1/\tau_r$. Given N packets to transmit (N large), the slowdown is

$$\begin{aligned} slowdown &= \frac{N\psi_c}{N/\tau_r} \\ &= (\tau_r/\tau_c)\left(\frac{\pi_{(1,2)}}{1-\pi_{(0,0)}-\pi_{(1,0)}}\right). \end{aligned}$$

We now use this model to investigate how different model parameters affect slowdown.

It is intuitive that slowdown will be impacted significantly by the availability of cover traffic—more and more cover traffic yields smaller slowdowns. The experiments we performed use cover traffic utilization as the independent variable. Another important factor is the relative sizes of cover and real sessions. Figure 4.5 plots slowdown as a function of cover traffic utilization, given a baseline native real utilization of 10%. Experiments identical to these with increasing real utilization yield essentially identical slowdowns. We plot slowdown for a variety of mixes of p_r and p_c. Three features of this data stand out. The first is that slowdown is a convex decreasing function of cover session utilization. When utilization is high there is a lot of traffic to be tunneled—provided that the sizes of the real sessions are large relative to the cover sessions. For even when cover utilization is high, a short real session will have to wait for a long cover session to finish before the real session can start. This is seen in the second feature of the data; for a given cover utilization, slowdown is an increasing function

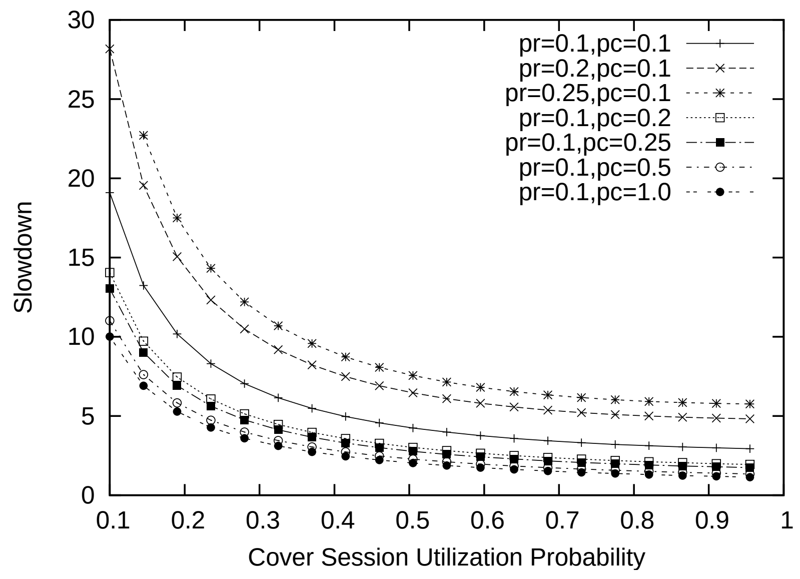


Figure 4.5: Slowdown as a function of cover traffic utilization when native real utilization is 10%

p_r/p_c (e.g., the ratio of the mean cover session size to mean real session size). The final interesting facet of the data is that even when p_r/p_c is advantageous, under low cover utilization the slowdown can be quite large. This reflects the waiting time a real session incurs for a cover session to come along and start to carry it.

The conclusions we can draw from this graph are that slowdown can be small—less than 25%, say, provided that one is willing to dedicate a large portion of bandwidth to the cover traffic relative to the real traffic, and that the cover sessions are short relative to real traffic sessions. Deviation from these conditions brings one into the realm of slowdowns that are a factor of 5 or more. One should note though that slowdown is a measure of the impact on *latency*. The conditions studied here all ensure that the tunneling does not impact the long-term rate of real traffic sessions.

4.3.3 Bandwidth

Next we consider bandwidth use. This is straightforward, appealing to the renewal structure of the session models. We view transmitted kilobytes as a reward; the rate at which this reward is earned is the mean consumed bandwidth. The theory of reward renewal processes says that if reward is earned at rate λ during an on cycle, and not at all during an off cycle, then the limiting rate at which the process as a whole earns reward is $\lambda E[\text{on time}]/(E[\text{on time}]+E[\text{off time}])$. Applied to our models, the cover session earns reward at rate $1/\tau_c$ during the on cycle, and the real session earns it at rate $1/\tau_r$ during the on cycle. The overall rate reward values are thus

$$\lambda_c = \left(\frac{\tau_c/p_c}{\tau_c/p_c+\mu_c}\right)\frac{1}{\tau_c},$$

and

$$\lambda_r = \left(\frac{\tau_r/p_r}{\tau_r/p_r+\mu_r}\right)\frac{1}{\tau_r}.$$

We have seen that the cover flow must be frequent enough to carry real flow without affecting throughput. An informative metric to compute is the ratio λ_c/λ_r, giving us the factor by which the cover traffic has more volume. We obtain

$$(\lambda_c/\lambda_r) = (p_r/p_c)\left(\frac{\tau_r/p_r+\mu_r}{\tau_c/p_c+\mu_c}\right).$$

This has the intuitive interpretation as the factor by which the cover traffic carries more packets during a session than does the real traffic, times the factor by which a native on-off cycle of the real traffic is longer than a cover traffic session. For fixed p_r, τ_r, and τ_c this ratio increases as $(1/p_c)$ increases, e.g., as the length of the cover session grows.

4.4 Conclusions

Independent cover traffic tunneling can exact a considerable performance cost to the real traffic being tunneled. In this chapter we investigate the properties of those performance costs using simulation and analytic modeling. We use the simulation to generate measurements of traffic behavior, using high fidelity models of protocols that carry traffic. We use the measurements to validate an analytic model of latency. We then augment the model to capture the dynamics of tunneling sessions, in a Markov chain formalism. From this model we can express conditions under which the real traffic throughput is unaffected by tunneling, and extract the slowdown—i.e., the factor by which the latency is increased by tunneling.

The models aid us in determining constraints to follow that ensure that the real traffic's throughput, latency or stability are bounded. As intuition suggests, real traffic slowdown is minimized when there is ample cover traffic to carry it. Similarly, real bandwidth grows as the volume of cover traffic grows in relation to the real traffic volume. However, we found that mismatches between the relative sizes of cover and real sessions can result in significant performance degradation. These mismatches can result from having a cover stream that is too small, and though its utilization is high, the inter session time can drastically increase latency. When the cover stream is too large, the real traffic has to wait for large empty cover streams to complete before real transmission can begin. Even when the relative sizes are favorable, a cover session model with low utilization can result in higher slowdowns. Thus, the waiting times that arise from the combination of real and cover traffic have a greater impact on performance than padding or native network transmission time.

We found that the nature of our problem allows for an interesting and useful decoupling of models. We show that it is possible to take measurements from real or simulated traffic and use it to estimate parameters for an accurate analytic model of tunneling, *without the measurements or simulations having any notion at all of tunneling*. Analysis of the analytic model shows that its predictions of slowdown are very dependent on the accuracy of the ratios of the mean time to transmit a packet using the cover session pattern, to the mean time to transmit it in the native real flow. The analytic model is useful as an explanatory device without accurate measurements of these means, but is questionable as a *predictive* device without them. Together these different model forms can address analysis of tunneled traffic in a way that neither can individually.

CHAPTER 5

ENHANCING PERFORMANCE WITH BIASED COVER TRAFFIC

Existing encrypted traffic analysis attacks rely on some sort of inference technique to reconstruct information about the plaintext. Since these techniques have nonzero error rates, we have the ability to relax our independence assumption for cover traffic tunneling to improve performance. We approach this problem by biasing the selection of cover traffic parameters based on the current state of the real traffic. We treat the biasing process as a per-parameter black-box function, which transforms the parameter's distribution into an optimized selection.

We investigate biasing through simulation and Internet study of various functions we have developed using intuition from our previous simulation and analytic modeling work in Chapter 4. We focus on biasing on object size parameters (request and response sizes from Table 2.1).

The degree to which we can make cover traffic generation dependent on the real traffic to improve performance is security-sensitive. In this chapter, we investigate the following questions about biasing: How much does biasing aid reconstruction of the plaintext by an attacker? What impact does biasing have on defense detection? With knowledge of the expected protocol behavior of the real traffic, can a user optimally select a cover traffic model? How does biasing improve the performance of the TrafficMimic tests using an independent cover traffic model from Chapter 3?

5.1 Biasing

In the unoptimized system described in Chapter 2, TrafficMimic generates the stream of cover traffic *independently* from the data actually being sent by the real flow. This ensures that an attacker is unable to recover information leaked by the cover traffic. While secure, this scheme can produce poor performance when there is a mismatch between the traffic properties of the real and cover traffic. In this section, we describe methods for selecting cover traffic patterns with knowledge of the real traffic.

As described before, we collect empirical distributions of structural features learned from a real network trace using Swing. TrafficMimic uses distributions of request size, response size, inter-exchange time, and inter-connection time to generate cover traffic. To sample these distributions to create cover traffic, we perform the empirical equivalent of inverse transform sampling by converting uniform [0,1] variates to indices into the sorted array that stores the empirical distribution.

The general approach to our performance improvements involves biasing the selection of the samples from the empirical distribution. Rather than directly sampling, we call a function *bias* that takes the empirical distribution to sample, the state of the real flow, and a parameter that controls amount of biasing to apply. The state parameter describes the needs of the real flow for the traffic parameter described by the empirical distribution. This state can either be the instantaneous current value of a traffic property (e.g., the number of bytes of the real flow that are queued to be sent) or an estimate of the current state from history (e.g., the exponential moving average of past real inter-exchange times). This function returns a random sample from the empirical distribution whose selection is influenced by the current state of the real flow.

In this chapter, we focus on creating and evaluating bias functions that operate on the object size traffic properties (request or response sizes). TrafficMimic uses a local per-flow buffer of 64KB (q_{max}) to store pending real data at both the tm-server and tm-client. The bias functions use the amount of data in this pending buffer as their state parameter. We use the variable q to denote the amount of data in this buffer.

From our work in Chapter 4, we observed that frequent object splitting and the extra waiting it causes was the dominant reason for poor performance. We also found that excessive overage impacted latency because large underutilized cover objects cause the cover flow to stall sending padding when real data is waiting. We use these insights to design bias functions that avoid splitting and minimize excess bytes. Though these assumptions may not be optimal[1], for all protocols, we show that they can considerably improve performance for many common protocols.

5.1.1 Functional Biasing

The first class of biasing techniques work by influencing the selection of samples from the empirical distribution using a statistical distribution described by the probability density function f_b. f_b is a piecewise continuous distribution[2] that maps uniform [0,1] to biased [0,1] given the cumulative fraction x_0 of the current value of q. We find this value by using the empirical observations of the CDF F_{cov}.

$$F_{cov}(q) = x_0$$

We then randomly sample the distribution described by f_b using inverse transform sampling. This requires that we derived the inverse CDF F_b^{-1} of f_b. We also need a [0,1] uniform random variable y as input to this process. Thus, F_b^{-1} takes parameters y, x_0, and a set of parameters denoted by *param*.

$$F_b^{-1}(y, x_0, param) = x_s$$

Now we can select a biased sample c from the cover distribution using its inverse CDF and a biased [0,1] random variate x_s.

$$F_{cov}^{-1}(x_s) = c$$

We next describe several candidate f_b distributions that we will use to bias object size distributions.

[1]Avoiding splitting and minimizing excess bytes may not be optimal, especially for protocols (e.g., VoIP) that differ considerably from our test set of SSH, SMTP, and HTTPS.

[2]Note that f_b is continuous, even though f_{cov} could be discrete. We assume that f_b is continuous so that it can be applied to any traffic parameter (i.e., one that *is* continuous, like interconnection time). We utilize the natural binning that occurs with a sampled empirical distribution to convert between discrete and continuous representations.

Probability Split

We start with a simple bias distribution. It attempts to avoid object splitting by preferentially selecting values greater than q with a fixed probability $1-p$.

$$f_b(x,x_0,p) = \begin{cases} \frac{p}{x_0} & 0 \le x < x_0 \\ \frac{1-p}{1-x_0} & x_0 \le x \le 1 \end{cases}$$

We derive the CDF by integration:

$$\int_0^x \frac{p}{x_0} dz = \frac{px}{x_0}$$

$$\int_0^{x_0} \frac{p}{x_0} dz + \int_{x_0}^{x} \frac{1-p}{1-x_0} dz = p + \left(\frac{1-p}{1-x_0}\right)(x-x_0)$$

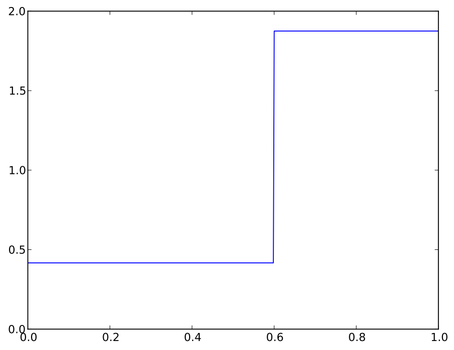
$$F_b(x,x_0,p) = \begin{cases} \frac{px}{x_0} & 0 \le x < x_0 \\ p + \left(\frac{1-p}{1-x_0}\right)(x-x_0) & x_0 \le x \le 1 \end{cases}$$

and invert the CDF to yield F_b^{-1}

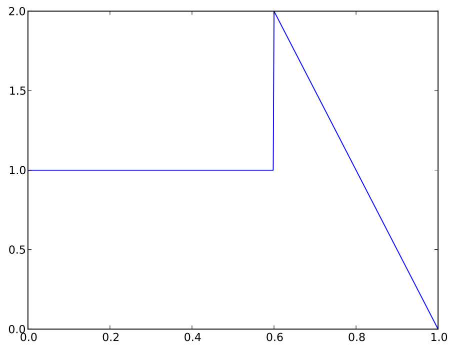
$$F_b^{-1}(y,x_0,p) = \begin{cases} \frac{yx_0}{p} & 0 \le y < p \\ x_0 + \frac{(y-p)(1-x_0)}{1-p} & p \le y \le 1 \end{cases}$$

Since the bias functions that we develop have different parameter ranges, we develop a standardized parameter α that increases bias as the parameter grows larger. In all cases, we consider $\alpha = 0$ to designate no biasing. For this probability distribution, we consider α to be related to p as follows: $p = 1/\alpha$. Figure 5.1(a) shows a sample of the PDF of this function graphically where $p = 0.25$ (similarly $\alpha = 4$).

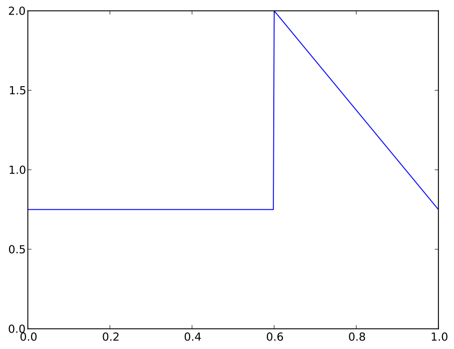
Figure 5.1: Sample PDFs of various bias functions with $x_0 = 0.6$



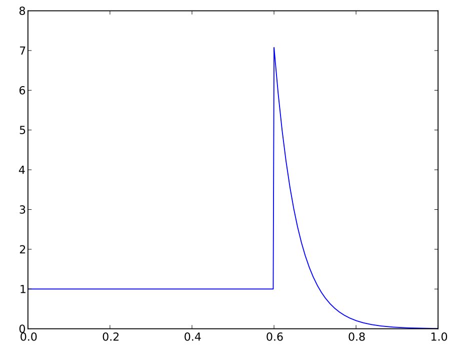
(a) Probability split



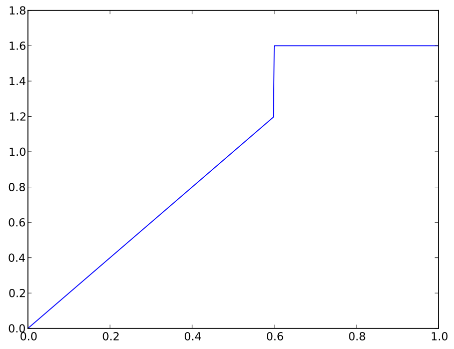
(b) Linear decay



(c) Linear decay variation



(d) Exponential decay



(e) Reverse linear

Linear Decay

The next distribution we describe features a uniform probability below x_0 and then a linear decay for $x \geq x_0$. The motivation for this function is to bias giving mass larger than x_0 while simultaneously limiting overhead. To construct a PDF for this distribution we construct a piesewise function. The height of the distribution is h for $x < x_0$. For $x_0 \geq x$, we have a triangle with area $1 - x_0 h$. We can then find its height y_0 (i.e., the height of the optimal point x_0). We then find the slope of the line forming the triangle and its intercept.

$$f_b(x, x_0, h) = \begin{cases} h & 0 \leq x < x_0 \\ \frac{2(hx_0-1)(x-1)}{(x_0-1)^2} & x_0 \leq x \leq 1 \end{cases}$$

$$F_b(x, x_0, h) = \begin{cases} hx & 0 \leq x < x_0 \\ 1 + (hx_0 - 1)\frac{(x-1)^2}{(x_0-1)^2} & x_0 \leq x \leq 1 \end{cases}$$

We can then use the inverse of the CDF to create biased samples from the cover size distribution.

$$F_b^{-1}(y, x_0, h) = \begin{cases} \frac{y}{h} & 0 \leq y < hx_0 \\ 1 + (x_0 - 1)\sqrt{\frac{(y-1)}{(hx_0-1)}} & hx_0 \leq y \leq 1 \end{cases}$$

Again we use parameter α, where $h = 1/\alpha$. Figure 5.1(b) shows a sample of this function graphically where $h = 1$.

Linear Decay Variation

We also created a variation of the above linear decay function, which decays to h rather than 0 on the right side. We use a similar to technique to derive the PDF, CDF, and inverse CDF.

$$f_b(x,x_0,h) = \begin{cases} h & 0 \le x < x_0 \\ \frac{2x(h-1)}{(x_0-1)^2} + \frac{\left(h(x_0^2-2x_0-1)+2\right)}{(x_0-1)^2} & x_0 \le x \le 1 \end{cases}$$

$$F_b(x,x_0,h) = \begin{cases} hx & 0 \le x < x_0 \\ \frac{((h-1)x^2+(h(x_0^2-2x_0-1)+2)x-(h(x_0^2-x_0-1)-x_0+2)}{(x_0-1)^2} + hx_0 & x_0 \le x \le 1 \end{cases}$$

$$F_b^{-1}(y,x_0,h) = \begin{cases} \frac{y}{h} & 0 \le y < hx_0 \\ \frac{|x_0-1|\sqrt{4(h-1)y+h^2(x_0^2-2x_0+1)-4h+4}-h(x_0^2-2x_0-1)-2)}{2(h-1)} & hx_0 \le y \le 1 \end{cases}$$

We use the same input parameter standardization: $h = {}^{1}\!/\!_{\alpha}$. Figure 5.1(c) shows a sample of this variation graphically where $h = 1$.

Exponential Decay

Rather than linear decay after x_0 we can use exponential decay.

$$pdf(x,x_0,h) = \begin{cases} h & 0 \le x < x_0 \\ e^{b-mx} & x_0 \le x \le 1 \end{cases}$$

$$1 - hx_0 = \int_{x_0}^{1} e^{b-mx} dx$$

$$e^b = \frac{(1-hx_0)m}{e^{-mx_0} - e^{-m}}$$

Because we want the exponential to have a finite tail, we simplify by considering the term $e^{-m} = 0$. Thus:

$$b \approx log\left(m(1-hx_0)\right) + amx$$

$$f_b(x,x_0,h) = \begin{cases} h & 0 \leq x < x_0 \\ m(1-hx_0) \cdot e^{m(x_0-x)} & x_0 \leq x \leq 1 \end{cases}$$

We next need to parameterize this function by selecting an appropriate m given a value of x_0. We choose to limit the error caused by the simplification above. We use ε to represent the error factor at $x = 1$

$$\varepsilon = e^{-m} \approx 0$$

We parameterize ε by β as follows:

$$\beta = -log(\varepsilon)$$

Thus, m for a given β and x_0 is

$$m = \frac{log(1-x_0)+\beta}{1-x_0}$$

We use a static value of β and use h to vary the biasing level of the function. Next to convert to a CDF, we integrate

$$F_b(x,x_0h) = \begin{cases} hx & 0 \leq x < x_0 \\ hx_0 + (1-x_0)\left(1-e^{m(x_0-x)}\right) & x_0 \leq x \leq 1 \end{cases}$$

$$F_b^{-1}(x,x_0,h) = \begin{cases} \frac{y}{h} & 0 \leq y < hx_0 \\ x_0 - \frac{1}{m}log\left(1-\frac{y-hx_0}{1-hx_0}\right) & hx_0 \leq y < 1 \end{cases}$$

Again we use the standardization $h = 1/\alpha$. Figure 5.1(d) shows a sample of the exponential decay bias distribution graphically where $h = 0.75$.

Reverse Linear

We next evaluate a distribution that attempts to cause fewer splits even when some splits will occur. The intuition is that sending larger cover sizes even when causing splits, reduces the total number of splits. To accomplish this, we create another piecewise function. Unlike the previous functions, this distribution has a flat probability density to the right of x_0. It uses a linear increasing function to the left of x_0. The height of the triangle is a fraction p of the height of the flat line to the right of x_0. Using this technique and parameterization we can create a PDF, CDF, and inverse CDF. First we can find h using p:

$$h = \frac{2}{px_0 - 2(x_0 - 1)x_0}$$

The remainder of the derivation follows as before:

$$f_b(x, x_0, p) = \begin{cases} \frac{2px}{(px_0 - 2(x_0-1))x_0} & 0 \leq x < x_0 \\ h & x_0 \leq x \leq 1 \end{cases}$$

$$F_b(x, x_0, p) = \begin{cases} \frac{hpx^2}{2x_0} & 0 \leq x < x_0 \\ \frac{h(px^2 + 2x_0x - 2x_0^2)}{2x_0} & x_0 \leq x \leq 1 \end{cases}$$

$$F_b^{-1}(y, x_0, p) = \begin{cases} \sqrt{\frac{2x_0}{hp} y} & 0 \leq y < \frac{x_0 ph}{2} \\ \sqrt{x_0} \frac{\sqrt{2pyh(2p+1)x_0}}{\sqrt{hp}} - \frac{x_0}{p} & \frac{x_0 ph}{2} \leq y \leq 1 \end{cases}$$

Because this function's parameter is related to the area to the left of x_0, we use a different parameter standardization: $p = 1 - {}^1/_\alpha$. Figure 5.1(e) shows a sample of this function graphically where $p = 0.75$.

5.1.2 Algorithmic Biasing

The next class bias techniques we investigate work by selecting biased samples using an algorithm tailored to the traffic property being biased. These functions work directly towards our performance goals rather than indirectly through a statistical distribution. Each algorithm samples the empirical cover distribution several times until it finds a suitable value. We call this basic approach Try, Try Again (TTA).

Linear TTA

This algorithm works by iteratively sampling the cover distribution up to r times. The first sample c_i, where $c_i > q$, is used to transmit the data. If no suitable c is found after r samples, the final sample c_r is used. Figure 5.2(b) shows a sample output distribution from this algorithm compared to an unbiased Geometric in Figure 5.2(a).

Optimal TTA

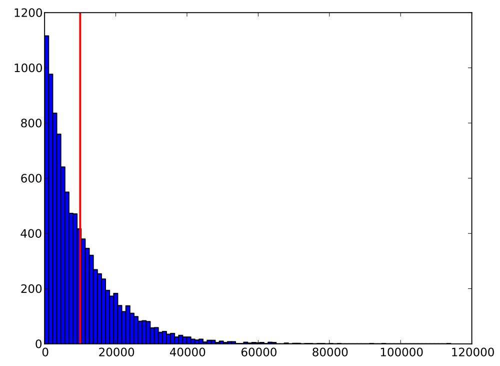
We also developed an enhancement to this algorithm that attempts to find "best fit" c given the amount data in the pending buffer. Given a set of r random samples $c_1, \ldots, c_r$ and q bytes in the pending buffer, the algorithm selects c such that $c > q$ and c is the minimum of all $c_i > q$. If there is no $c_i > q$, then the algorithm selects c to be the maximum over all c_i. Figure 5.2(c) shows a sample output distribution from this algorithm.

5.1.3 Sampling Without Replacement

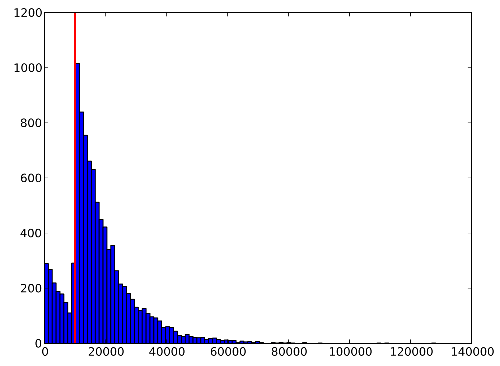
A final consideration for each of the bias functions we have described thus far is the sampling method. To preserve the shape of the expected cover size distribution to prevent defense detection attacks, we can sample without replacement. To prevent exhaustion of the empirical data, we first create a L length subset of samples from the empirical distribution. We then can run the above algorithms on this subset and sample without replacement. When exhausted, the algorithm regenerates a new subset of length L. This ensures that the shape of the observed cover size distribution remains consistent within a window of L observations.

Figure 5.2: Illustration of algorithmic biasing of a Geometric distribution (mean 9.77e-5) with $r = 4$ and $q = 10000$ (q shown in red)

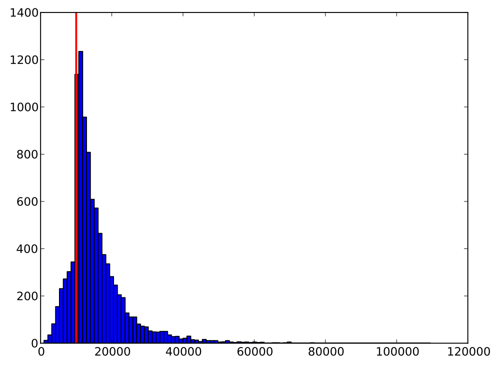
(a) Unbiased



(b) Biased with linear algorithm



(c) Biased with optimal algorithm



We can sample without replacement with both functional and algorithmic biasing. We treat the subset F_{sub} like the original ECDF, F_{cov}. The functional techniques use the biased $[0,1]$ value to index into F_{sub}. Similarly, the algorithmic techniques select r samples from F_{sub} before selecting a biased cover size.

5.2 Attacks

We now present the attack and evaluation methodology we use to quantify and compare the differences between biasing methods. We use simulation to evaluate the attack because the low-noise of the simulation favors the attacker. We use information theoretic metrics to compare information leaked by biasing.

5.2.1 Bayesian Attack

We use Bayesian data analysis to form an attack against biasing. The goal of the attacker is to infer the value of q given observation of a single cover session size cov_i traversing the network. We present this attack as a theoretical examination of how much information the attacker can recover when TrafficMimic is biased. We neither expect this attack perform exceptionally well in practice nor do we provide the design and evaluation of an attack that can recover the original object size given a set of inferred q values.

We use Bayes Theorem as follows:

$$P(q|cov_i) = \frac{p(cov_i|q)p(q)}{p(cov_i)} \tag{5.1}$$

We assume that the attacker knows the distribution of queue and cover session sizes. We further assume that the attacker has full details of TrafficMimic's operation and biasing algorithms (including q_{max} and the biasing parameter α or r).

In the following sections we describe the three phases of this attack: pre-computation, observation, and attack.

1. **Pre-computation** Before conducting the attack, the attacker needs to compute the values of $p(cov|q)$ for each possible value of q given a specific biasing algorithm and parameters. With functional biasing, the attacker can compute these probabilities using the PDF and CDF of the bias distribution. For algorithmic techniques, the attacker creates a distribution of probabilities by sampling the output of the bias algorithm to create an empirical distribution of `distsize` elements. To make these computation scalable, we allow the attacker to bin the q sizes using the `bufincrement` variable. We also perform smoothing using a kernel density estimator [57] on the distributions to create empirical probability mass functions that can be later used in the Bayes calculations.

2. **Observation** In the next phase, the attacker observes cover sizes on the network that he/she wishes to attack. In our attack, we use a simple simulator to generate data for this phase. The simulator first samples a real session size to transfer using TrafficMimic. It creates a buffer of size q_{max} and automatically fills it as space and real data are available. It then uses the same logic as TrafficMimic to bias and select cover sizes to cover the real traffic. The simulator runs many iterations of real flows (q_{sim_i}) which each generate one or more cover sizes (cov_i). The simulator stores the results in a tuple (q_{sim_i}, cov_i), which is passed to the next phase. The simulator also computes the average overage (this is perfect overage with no TrafficMimic overhead, only excess padding bytes sent) and average number of cover sessions required to send a real object. These values are proxies for the real performance of the biasing technique when used with TrafficMimic.

3. **Attack Application** In this phase, the attacker takes an observed cover size from the simulation, cov_i, and uses equation 5.1 to generate a probability distribution of $P(q|cov_i)$ for all possible q_i. The attacker chooses a normalizing factor, $p(cov_i)$, for equation 5.1 that ensures that all probabilities sum to one over a sufficiently large sample set. Empirically estimating this value is difficult because of sampling error. Next the attacker smooths this distribution with a kernel density estimator and finds the maximum $P(q_i|cov)$. We call this maximal likelihood

object size q_{est_i}. The attacker repeats this process for all cov_i.

The attacker can then take the q_{sim_i} and compare it to q_{est_i} to test the accuracy of the attack. We consider the attack successful if q_{est_i} is within the attack bin width (i.e., `bufincrement`) of q_{sim_i}.

5.2.2 Mutual Information

In addition to the raw attack accuracy described above, we use several mutual information metrics to test the attack. First we define some notation. We use Q_{sim} to denote a random variable from the distribution observed in q_{sim}. Likewise, we use Q_{est} to denote a random variable from the distribution observed in q_{est}. Similarly, we use C_{sim} to denote a random variable for distribution cov_i.

To directly evaluate the information recovered using the Bayesian attack, we compute the mutual information of Q_{sim} between Q_{est}:

$$I(Q_{sim};Q_{est}) = \sum_{q_{sim}\in Q_{sim}} \sum_{q_{est}\in Q_{est}} p(q_{sim},q_{est}) \log\left(\frac{p(q_{sim},q_{est})}{p(q_{sim})\,p(q_{est})}\right) \tag{5.2}$$

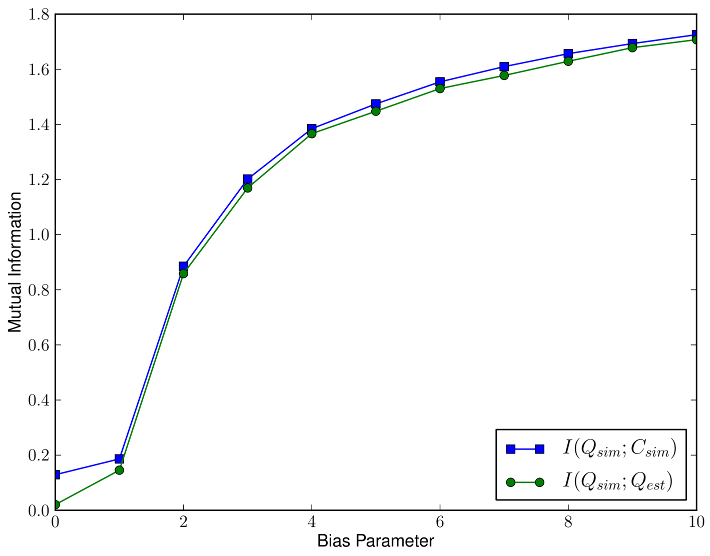
This requires that we compute the joint probability distribution function for Q_{sim} and Q_{est}. We use the methods described by Ince et al. to compute this mutual information [58]. We use the bin width estimator from Silverman [57].

We also develop an estimator for $I(Q_{sim};Q_{est})$ using only C_{sim}. Bayes rule provides a result for Q_{est} by taking the following input: priors of C_{sim} and Q_{sim} and the conditional probability of C_{sim} given Q_{sim} (i.e., $p(cov_i|q)$ from equation 5.1). We use M to denote the random variable from the distribution of $p(cov_i|q)$. We can then consider Bayes rules as a function f as follows:

$$Q_{est} = f(Q_{sim},M,C_{sim})$$

Consider a simplified estimation function g, which takes only C_{sim} as input and produces an estimate of q_g (and its associated random variable Q_g):

Figure 5.3: Mutual information estimator comparison for linearBias with HTTPS traffic



$$q_g = g(C_{sim})$$

The prior distribution of Q_{sim} is fixed and does not contribute additional information to Q_{est}. Furthermore, there is no information in M that is not contained in C_{sim}. Which leads to the following approximation:

$$I(Q_{sim};Q_{est}) \approx I(Q_{sim};Q_g)$$

The data processing inequality bounds the amount of information in Q_g to that in C_{sim}:

$$I(Q_{sim};Q_g) \geq I(Q_{sim};C_{sim})$$

Under reasonable conditions[3] on g, we find:

$$I(Q_{sim};Q_{est}) \approx I(Q_{sim};C_{sim})$$

[3]In the case that g is a sufficient statistic, then $I(Q_{sim};Q_g) = I(Q_{sim};C_{sim})$

To test this estimator, we computed both mutual information metrics from simulation runs collected from each of the bias functions described above. Figure 5.3 shows the similarity between mutual information computed using $I(Q_{sim};Q_{est})$ and $I(Q_{sim};C_{sim})$ for the linearBias function[4]. We see that the estimator both preserves the shape of the mutual information metric and is within a small margin of error. For the remainder of the results in this paper, we use $I(Q_{sim};C_{sim})$ because it is faster to compute.

5.3 Simulation Study

We again use network trace data from the CAIDA (for HTTPS and SMTP) and UNC (for SSH). In all results, we take 20,000 real objects and simulate sending them through a tunnel of cover traffic with each of the biasing functions. As in Chapter 4, we use the notation *realprotocol* / *coverprotocol* for specifying combinations of real and cover protocol models. Unless otherwise specified, we use *response* size distributions.

For comparison to existing constant padding techniques, we also test a biasing function that always selects a fixed size object (constantMTU). We tested a range of ten fixed object sizes between the lower and upper quartiles of each real distribution. This range of fixed object sizes illustrate some of the trade-offs between overhead and number of sessions for a constant rate scheme.

5.3.1 Performance

We first evaluate biasing HTTPS real traffic from the February CAIDA trace with HTTPS cover traffic learned from January. We use the algorithms with replacement described in Section 5.1. Figures 5.4 through 5.7 show the results of this experiment. Bias parameter 0 in all the series corresponds to unbiased TrafficMimic performance.

As expected, the biasing algorithms are able to reduce the average number of cover sessions required to send the real object. Since overage in our simulation is only a reflection of the last cover size required to send each real object, we observed larger

[4]linearBias is representative of the estimator's performance for other bias techniques

Figure 5.4: Session performance of HTTPS/HTTPS biased with replacement

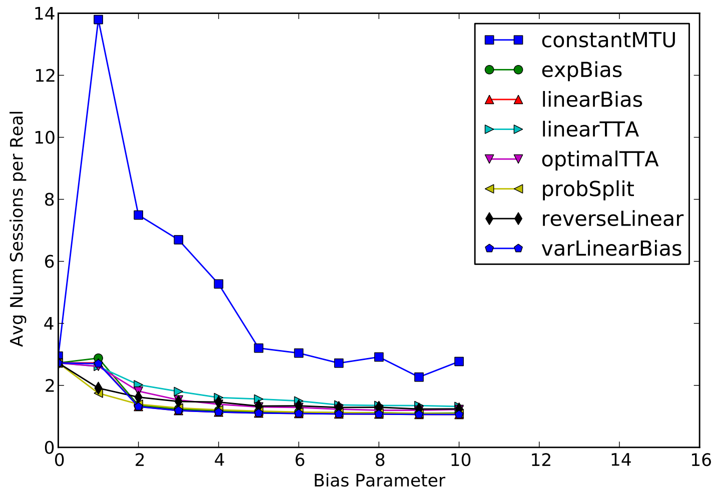


Figure 5.5: Overhead of HTTPS/HTTPS biased with replacement

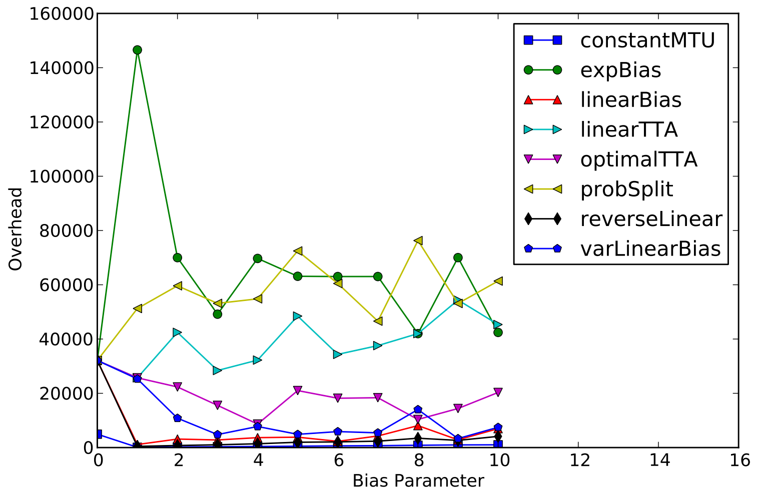
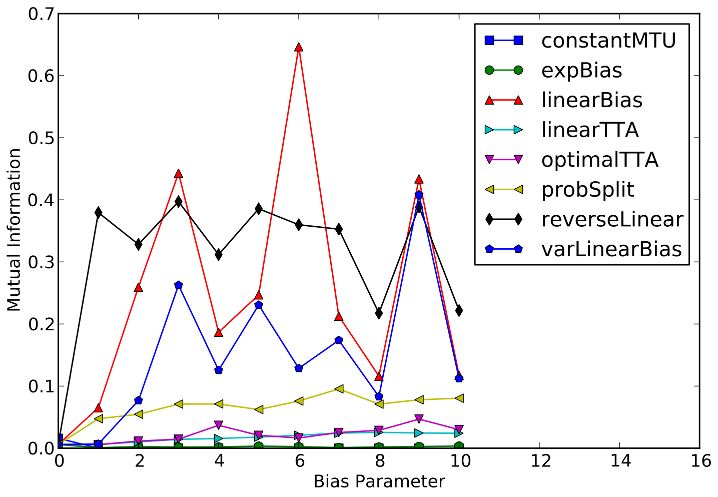


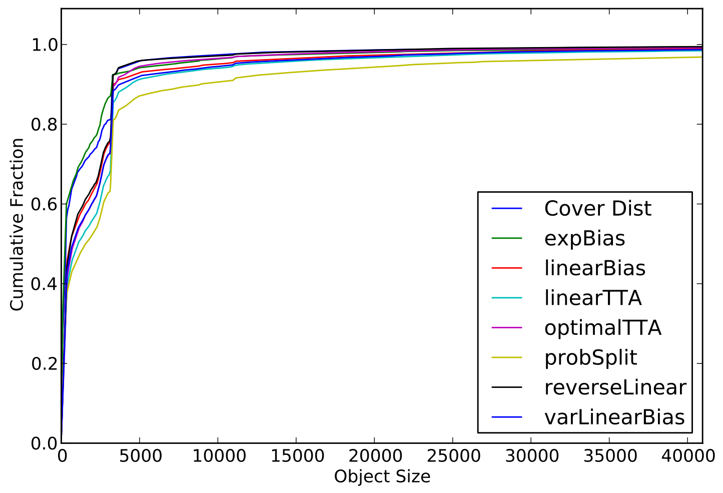
Figure 5.6: Mutual information of HTTPS/HTTPS biased with replacement



than expected variations across bias techniques. This will be evaluated in more detail in section 5.5. As Figure 5.5 shows, several algorithms can reduce overage, others (e.g. linearTTA, expBias, and probSplit) increase overage while still providing good session performance. This small additional overage is unlikely to affect HTTPS performance considerably since it will usually add only one additional object transmission time to the user-perceived latency. Figure 5.6 shows that linear functional techniques leak the most information about the real object sizes while the algorithmic methods (linear and optimal TTA) and expBias leak the least information. We observed a small amount mutual information with no biasing because both distributions drawn from the same underlying protocol HTTPS.

Since all of these tests are performed with replacement, the biased cover distributions are skewed from the original expected cover distribution. Figure 5.7 shows a zoomed view of the empirical CDFs for each biasing algorithm across the entire simulation for bias parameter 4. We use the two-sample Kolmogorov-Smirnov test to investigate the relative differences between the biased cover distributions. All the bias algorithms have a zero probability of being drawn from the unbiased cover distribution based on

Figure 5.7: HTTPS/HTTPS observed cover size distribution



the confidence estimation given by Stephens [59, 60] for bias factors greater than 1. So, both constantMTU and sampling with replacement provide minimal resistance to defense detection. We investigate the practicality of using KS as a defense detection attack in more detail in Section 5.3.3.

5.3.2 Traffic Combinations

We next simulated the SSH protocol over SMTP (See Figures 5.8 through 5.10). In our previous study, we found that both SMTP and SSH are dissimilarly asymmetric in the number of bytes sent and received. Because of this, we found more striking differences between biasing algorithms. The functional algorithms that favored mass above x_0, were able to reduce the number of sessions to nearly 1. However, these algorithms also resulted in the most overage and considerable information leakage (over 1 bit per cover object). This is due to SSH needing to send a large response but SMTP only offering smaller cover sizes. When this distribution is skewed by the bias functions, they push the SMTP protocol model into a less common but still possible state where its bytes sent versus received ratio is reversed.

Figure 5.8: Session performance of SSH/SMTP biased with replacement

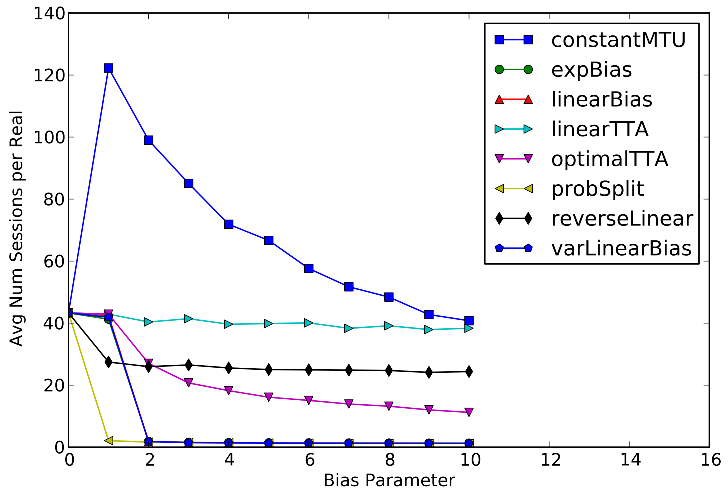


Figure 5.9: Mutual information of SSH/SMTP biased with replacement

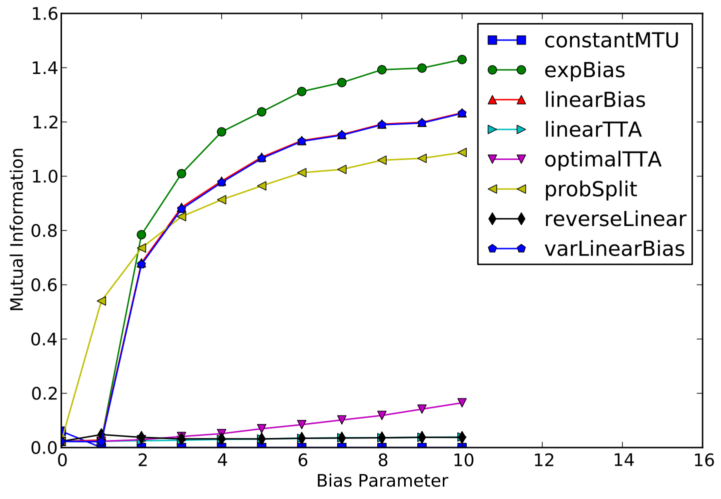
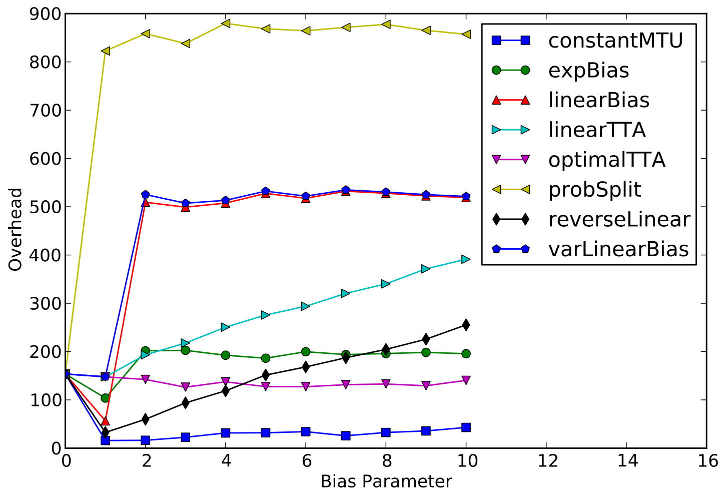


Figure 5.10: Overhead of SSH/SMTP biased with replacement



The algorithmic approaches and reverseLinear were less sensitive to this problem. However, they were only able to provide modest performance advantages. This is because these algorithms would only occasionally reverse the bytes sent ratio as evidenced by the very small overhead observed in Figure 5.10.

We next looked at sending HTTPS requests over a cover traffic stream of HTTPS responses. In this case, the unbiased cover distribution would usually be able to transmit the request in a single session. So, we consider the cover stream over provisioned for carrying the real stream. Thus, we observed that biasing had only a minimal effect on the session performance (see Figure 5.11). We also found in Figure 5.12 that several of the biasing techniques (e.g., optimalTTA, linearBias, reverseLinear, and varLinearBias) were able to reduce overage even though they are not specifically tuned for this over provisioned situation.

5.3.3 Defense Detection

We have already established that biasing with replacement provides minimal resistance to defense detection. We next want to understand how to reduce defense detection risk

Figure 5.11: Session performance of HTTPS-req/HTTPS-resp biased with replacement

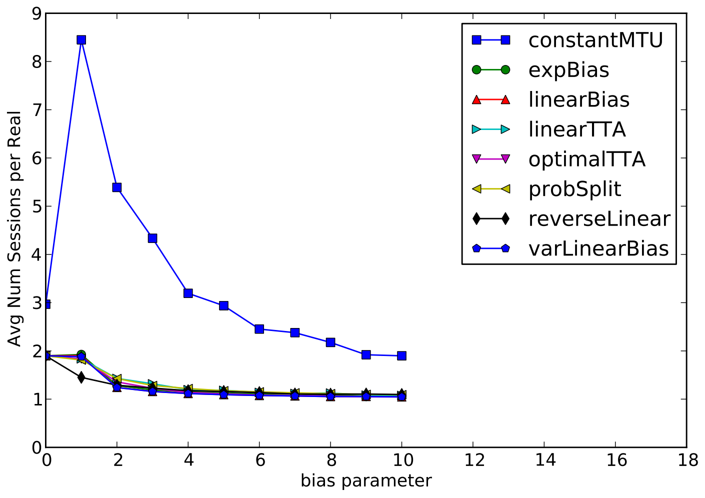


Figure 5.12: Overhead of HTTPS-req/HTTPS-resp biased with replacement

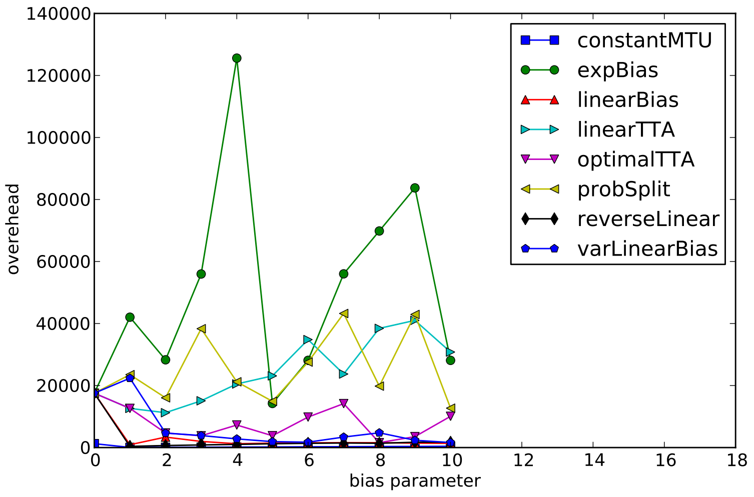
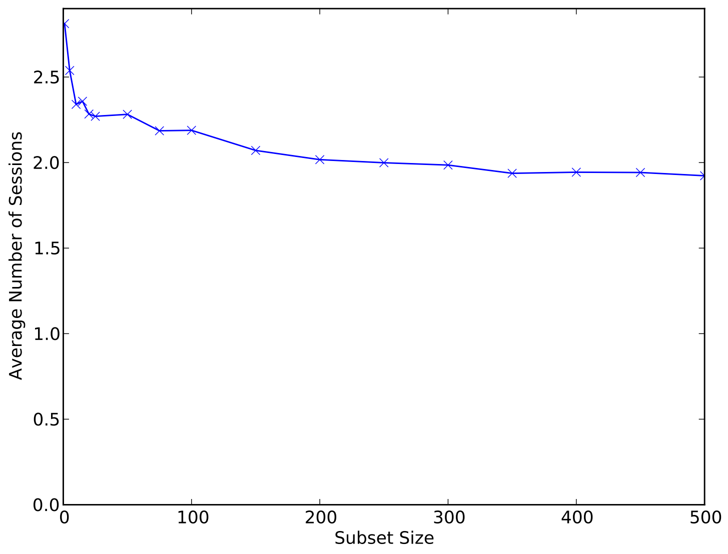


Figure 5.13: OptimalTTA without replacement and varying L sized subsets

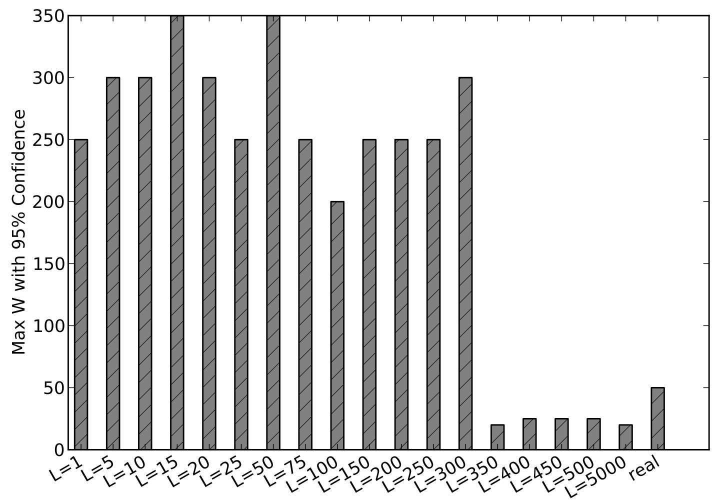


while retaining a session and overhead advantage over constantMTU. We approach this problem by sampling without replacement on a random subset of size L. This allows the biased cover distribution to remain consistent with the original distribution over a window of size L. We wish to investigate how varying the size of the sampling subset L impacts the attacker's ability to detect biasing using a KS-test.

In our previous examination in Section 5.3.1, we used data from the entire simulation of 20,000 real object transmissions to perform the KS-test. Realistically, the attacker should choose some smaller window W and test subsets of length W from the observed cover traffic. This allows the attacker to make a determination about defense detection much more quickly. The attacker faces a trade-off between noise impacting detection confidence and the amount of cover traffic needed to make a determination. Similarly, the defender has a trade-off between performance gain and low detectability. Intuitively, we expect the attacker to gain the detection advantage when $L > W$.

We investigated these competing trade-offs using several simulations. We first examine the session performance of choosing various values for L. We tested the optimalTTA algorithm on HTTPS traffic with parameter 5 with a variety of subset sizes.

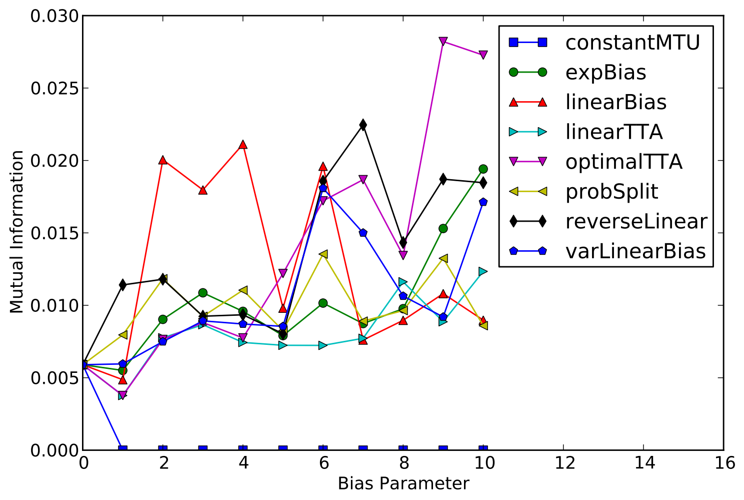
Figure 5.14: Windowed KS test attack on optimalTTA



We show the 99th percentile of session lengths in Figure 5.13. We see performance improvement over no biasing even with very small values of L (e.g., 5 or 10). We also tested a much larger subset length of 5000 and found the performance improvement remained modest at 7.8% fewer sessions than $L = 500$.

To test the attacker's trade-off on selecting a W, we use the two-sample KS-test on the original cover distribution (all the HTTPS response traffic from CAIDA February) compared to sampled biased cover distributions from Figure 5.13 at a variety of window and subset lengths. To understand the impact of noise, we also test the original cover distribution against an independent test set of unbiased data (learned from HTTPS traffic from CAIDA January). We used 95% confidence for the KS test to indicate that the observed samples are drawn from the same distribution. We repeated testing L length subsets for 100 iterations for each W. We show the maximum W for which each value of L falls within the attacker's confidence interval in Figure 5.14. We also show this for our independent set of HTTPS traffic (labeled 'real' in the chart). We found that windows above 50 reduced the confidence for the attack against the *real* test set. Furthermore, we found that the we were able to use values of L smaller than the window

Figure 5.15: Mutual information of HTTP/HTTP biased *without* replacement $L = 100$

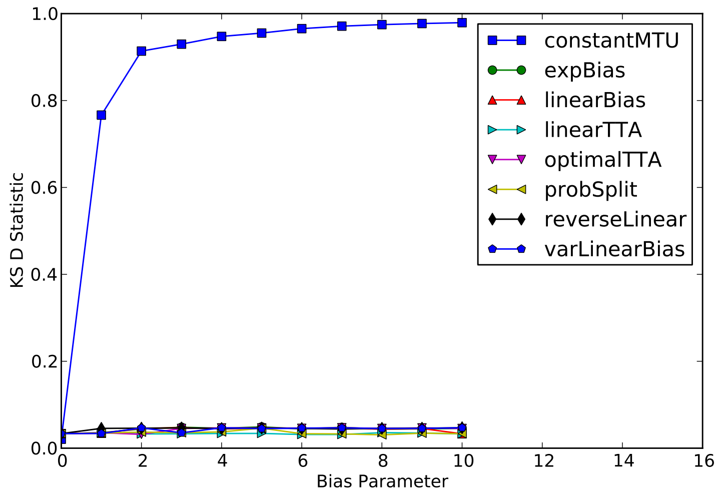


required by the attacker to detect it (deviating from our intuition about detection advantage above). To remain within the noise threshold for an independent test on real traffic, the attacker needs to use values of W 50 or smaller. So, we can safely use subsets of length 300 for this particular scenario. We use a conservative estimate of $L = 100$ for our remaining examination of sampling strategies.

We next wanted to investigate in more detail the performance properties of all our biasing techniques when sampled with replacement on a random subset of size 100. Figures 5.15 and 5.16 show the results of this test. As expected, the amount of information leakage when sampling without replacement was reduced from the what we observed in Figure 5.6. We see that the KS-statistics for the bias functions are now near or below the value we observed for bias factor 1 when sampling with replacement. This results in much higher confidence that the distributions are indistinguishable from real cover traffic.

Despite remaining consistent with the original cover distribution within the attack window, sampling without replacement still results in some performance gain. This is due to cover/real size matches that occur before the subset becomes depleted. Fig-

Figure 5.16: KS D statistic of HTTP/HTTP biased *without* replacement $L = 100$

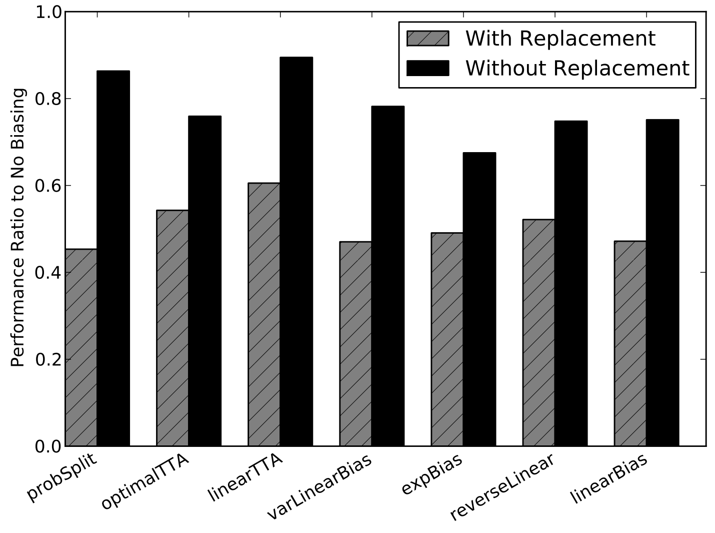


ure 5.17 shows the ratio of number of sessions with biasing to the number of sessions required without biasing. On average there is a 55% increase in the average number of sessions when sampling without replacement. Unlike sampling with replacement, average overage for all the biasing techniques was lower than the overage with no biasing. The overage decrease was especially true of bias functions (e.g., expBias and probSplit) that strongly bias selecting values larger than x_0. As the subsets became more depleted, very large cover sizes were quickly pruned out even though the bias algorithm was favoring values in the tail of the distribution.

5.3.4 Trading Off Performance and Security

The simulations we have developed can help us to understand the performance and security trade-offs between techniques. Given the complexities of combining protocols in tunnels, these simulations themselves can have a great impact on the user's choice of cover protocol. To illustrate this by example, we take the HTTPS/HTTPS experiment we performed before and show trade-offs in Figure 5.18. We see that there is considerable range for improving session performance while maintaining low informa-

Figure 5.17: Session performance compared to no biasing



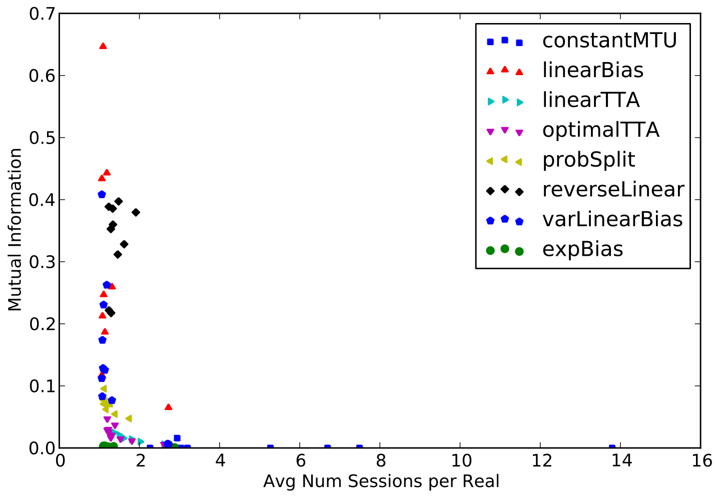
tion leakage (linearTTA, optimalTTA, expBias, and probSplit). However, expBias and probSplit can have higher overhead than the others. Also consider the example shown in reverseLinear. It provides good session performance and low overhead, but has one of the higher information leakages. Since these leaks are still small by comparison, this might prove appropriate for some user scenarios.

5.4 Biasing Implementation in TrafficMimic

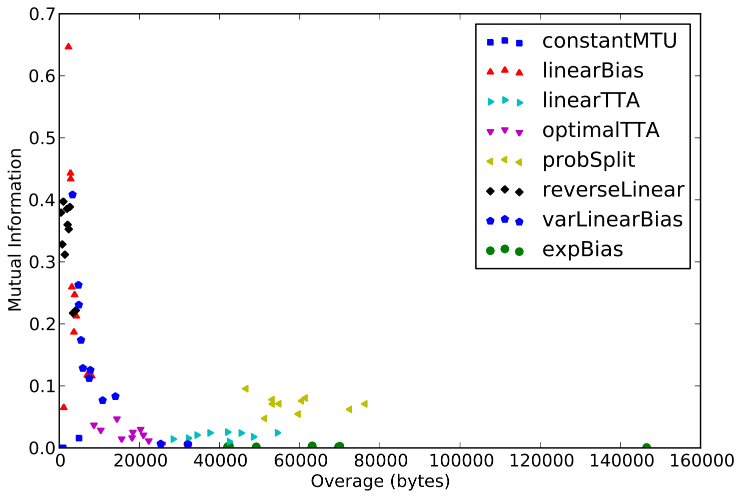
To better understand how these bias functions work on real networks and with a full cover traffic model (which includes timing, exchanges, and connections rather than uni-directional biased object sizes), we implemented algorithmic and functional biasing in our full C++ implementation described in Section 2.3. To enable biasing, we needed to change portions of the core traffic generation architecture. We also added a critical performance enhancement that we discovered during the development of our SSFNet simulator. We re-factored TrafficMimic to deliver real data as soon as it arrived rather

Figure 5.18: Performance and security trade-offs for HTTPS/HTTPS biasing

(a) Mutual information versus number of sessions



(b) Mutual information versus overage



than waiting for the entire cover traffic message that carries it to finish. This allows heavily padded messages to be delivered quickly and the destination to begin processing while the remainder of the padding finishes.

The primary change required for implementing biasing is that the biasing algorithms need the current state of the real traffic in order to select an optimal cover traffic parameter. We developed two solutions to this problem: feedback and split biasing.

5.4.1 Feedback

Since the master controls *all* cover traffic in a bidirectional cover traffic tunnel, this meant that we need a feedback mechanism to pass the current buffer state from the slave back through the tunnel to the master. Furthermore, the main event loop, where the state of the real connection is stored and serviced, runs in a separate thread from the master model thread. So, even the local real buffer state is not directly accessible to the master model thread (where biased parameters are selected). TrafficMimic already sends *confirmation* traffic_reqs back across the tunnel when responding to the master. The primary purpose of these confirmations is to synchronize the generation of traffic using IDs. We overloaded some unused fields in these confirmations to store the real buffer state. Upon receiving the confirmation, the master's event loop also embeds the local state into the confirmation. It then enqueues the confirmation to the master model thread. So, after sending some data that required a response, the master model thread will have an updated view of both the local and remote buffer states.

5.4.2 Split Biasing

We observed that when using algorithmic biasing, we could split the biasing process into two phases: one phase executed by the master model thread and one phase executed by the event loop. This allowed us to avoid having stale information about the local or remote buffer state because we can offload the selection of the optimal traffic parameter to the event loops of both the master and slave. Though this effect is minimal when applied to the local buffer state on the master, the potential for stale buffer information on the slave could be considerable. To split the biasing process we allow the master model thread to select *r* random variates from the appropriate distribution and encode them into a special type of traffic_req called a *traffic_req_multi*. Because small *r*s tend to produce considerable performance improvements (as we observed during simulation), the additional overhead is minimal. Unfortunately, this splitting can only be used with the TTA algorithms because they only need to send a small amount of data. Functional biasing methods would require transmission of large empirical distributions (thousands of observations) to offload biasing to the slave while still maintaining control of the biasing and model training processes at the master.

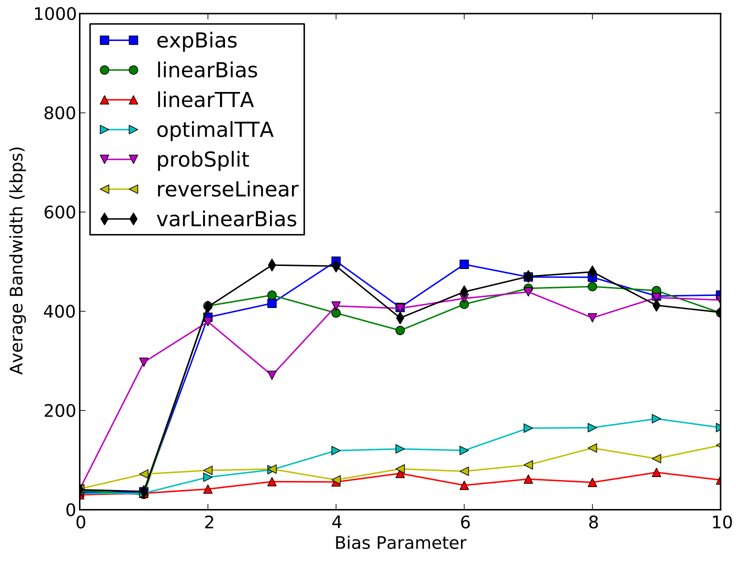
5.5 Real-World System Evaluation

Using our bias-enabled implementation of TrafficMimic we repeated some of the performance tests from Chapter 3. We used the MN-UK link for the tests. We trained Swing using the February CAIDA traces for HTTPS and SMTP and the 2003 UNC traces for SSH. We used each of the bias algorithms with replacement. Unless otherwise specified we used the feedback biasing implementation.

5.5.1 Bulk Transfer

We first repeated the 100KB bulk transfer using each of the biasing algorithms with a range of bias parameters. Figure 5.19 shows the results for transferring 100KB over the HTTPS response stream biased with each of the algorithms with replacement. We observed considerable increases in bandwidth, especially for functional biasing techniques. Bulk transferring 100KB (i.e., greater than q_{max}), ensures that x_0 remains nearly 1 for much of the test. This allows the functional algorithms to heavily utilize the long tails of the cover object size distributions. Since the algorithmic techniques have a relatively smaller working set of values from which to choose (i.e., maximum 10), their performance gain is more modest.

Figure 5.19: Bulk transfer over HTTPS-resp



We also show how biasing with optimalTTA improves the performance for different cover protocols in Figure 5.20. We perform a 100KB bulk transfer over the request stream of each protocol. We choose optimalTTA for its solid performance and low information leakage in a variety of scenarios as observed during simulation.

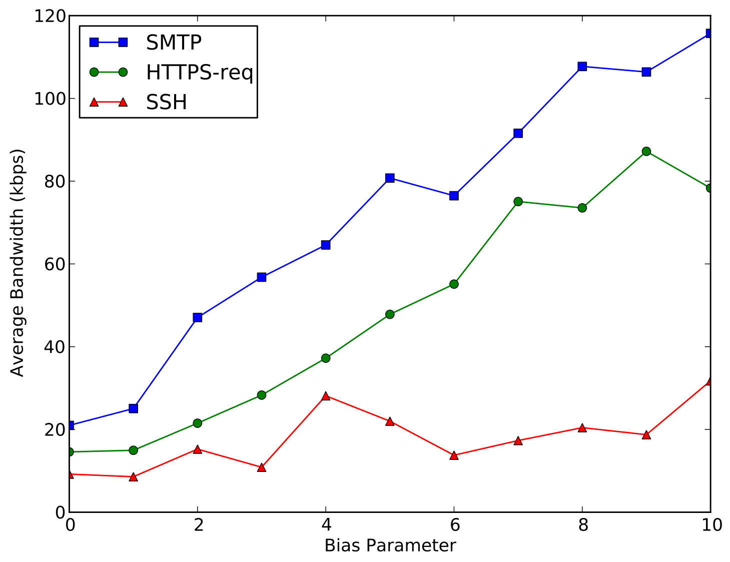
We saw reductions in *bidirectional* overhead for SSH and HTTPS-req. Since the SMTP request stream is larger than SSH and SMTP, there is minimal overhead due mostly to the need for an empty response stream. We observed a 3.5 to 5.5x improvement in bandwidth with biasing across all the algorithms. Unfortunately, biasing cannot improve the performance of SSH, without vastly changing its attacker observed output distribution thereby failing defense detection and leaking too much information. The optimalTTA algorithm prevents this problem at the expense of better performance.

5.5.2 Web Browsing

We next repeated the Web site load time experiment from Section 3.3.3. We show the results of biasing load times with the optimalTTA algorithm with parameter 5 in Figure 5.21. We show the speedup from using biasing compared to the unbiased data in

Figure 5.20: Bulk transfer with the optimalTTA algorithm

(a) Bandwidth



(b) Overhead

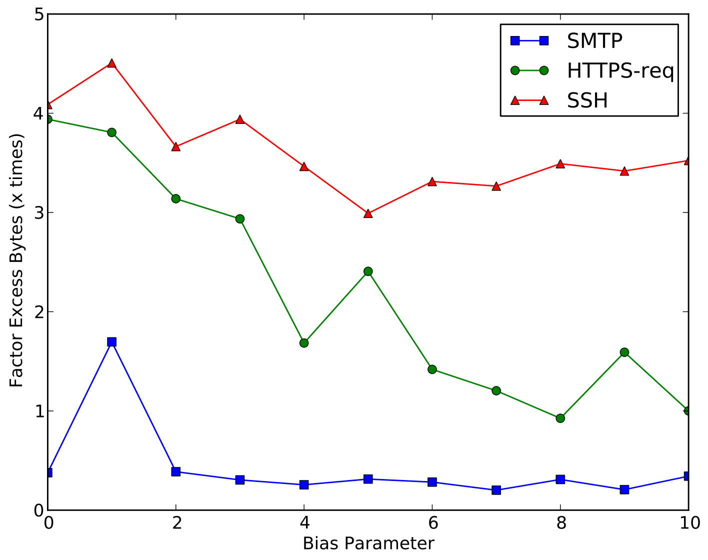


Table 5.1: Web browsing with biased cover traffic (seconds)

	HTTPS-Split	HTTPS	SMTP	SSH
google.com	6.45	6.42	72.53	14.77
facebook.com	8.82	11.65	42.44	11.47
youtube.com	14.62	15.18	126.10	36.64
yahoo.com	19.43	34.19	151.77	28.58
live.com	12.05	12.70	77.12	22.12

Figure 5.21: Website load time speedup with optimalTTA, parameter 5

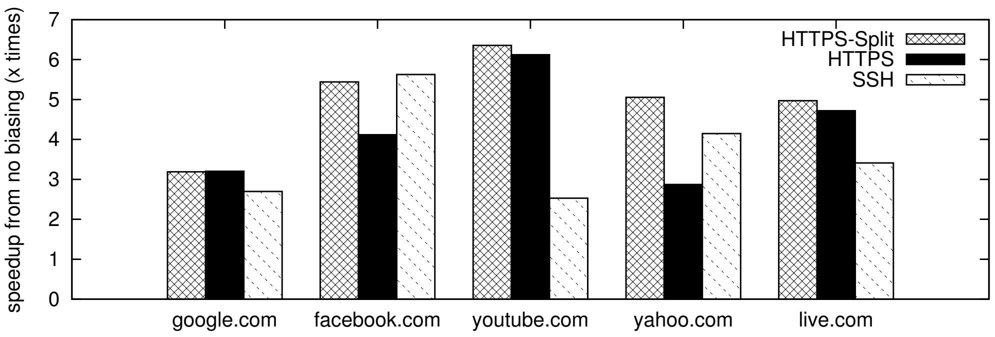
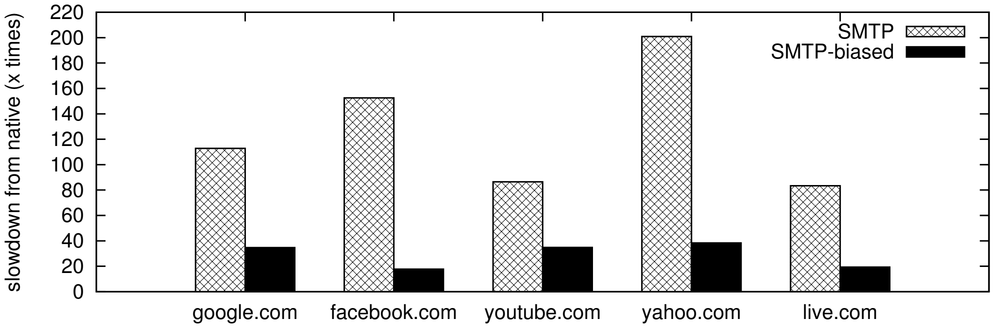


Figure 3.6. We show the raw values in Table 5.1. We found a 2.5-9.5x improvement in Website load time with biasing. We were also able to considerably improve the performance of Web browsing over the SMTP protocol (see Figure 5.22). Unfortunately, using this protocol for tunneling HTTP still falls short of practicality for most users. There are some fundamental incompatibilities in the interaction of these protocols. We found the most considerable problems with SMTP were related to long inter-exchange times and a relatively small response stream.

Lastly, we wanted to investigate the effect of feedback versus split biasing on performance. The Website test is most indicative of a real user because it includes both directions and some timing discrepancies. Thus, we are most likely to observe problems associated with stale buffer information in this test. We show the results of browsing to the top 5 websites with split biasing using HTTPS in Figure 5.21. We found that split biasing usually provided a small performance improvement over feedback. However, the degradation caused by feedback does not outweigh using functional biasing when

Figure 5.22: Web browsing over SMTP



it is more appropriate than algorithmic techniques.

5.6 Conclusions

In this chapter, we developed functional and algorithmic methods to bias the selection of cover traffic parameters based on the needs of the real traffic. Since we found splitting to have a significant impact on performance in Chapter 4, we performed an evaluation using object size distributions and biasing techniques that avoid splitting while minimizing overhead. We found that our biasing strategy was able to improve performance while bounding information leakage and retaining resistance to defense detection. Furthermore, the biasing techniques we have developed are general enough to be applied to other performance sensitive traffic parameters like packet or session timing.

We then performed a system evaluation of biasing in TrafficMimic using real Internet links. We confirmed the performance gains with biasing are considerable with real and synthetic workloads. Though cover traffic tunneling will always exact some performance cost, our work has shown that TrafficMimic can be used practically on the Internet. This helps to ensure that its integration into other privacy enhancing technologies like VPNs or anonymity systems is feasible for real users.

CHAPTER 6

RELATED WORK

In this chapter, we present related work in the research areas that this work spans. We also expand upon the related work on traffic analysis threats from Section 1.2.

6.1 Traffic Analysis

The traffic analysis attacks we described in Section 1.2 focus on attacking specific protocols usually at the connection level between two fixed hosts. There has also been prior work in analyzing cross-host connection behaviors. Karagiannis et al. developed a classification strategy called BLINC that uses host connection patterns, server/client traffic disparities, and community detection to perform flow classification [61]. They focus on peer-to-peer traffic, but find that their approach has value for detecting and understanding a wide variety of protocols. They also develop heuristics for detecting attacks and other types of anomalous behavior using traffic analysis.

Later work extended some of the concepts from BLINC to create traffic dispersion graphs (TDGs). These graphs model the social behavior of the network using a directed graph where edges are defined with some network interaction [62]. They construct TDGs using both cross-host connection behavior and single connection features (e.g., bytes sent/received). Iliofotou et al. go on to create dynamically updating TDGs and use them to detect polymorphic blending attacks [63]. They find that they are able to correctly classify peer-to-peer traffic attempting to blend with a set of standard protocols (e.g., HTTP, DNS, etc...) even when the amount of blended peer-to-peer traffic is only 10% of the standard traffic. TrafficMimic would be able to evade single connection statistics used by a TDG-based detector but does not obscure cross-host connection patterns without being used with a VPN or proxy.

Traffic analysis is an effective defensive tool for detecting the command and control signals from botnets. Botsniffer [64] used traffic analysis techniques to detect bonet command control, which relied on cross-host connection counting similar to the work on BLINC. Gu et al. later developed BotMiner, which more fully took advantage of traffic analysis for command and control detection [65]. BotMiner's c-plane monitor collected netflow records on which hosts were communicating and used non-parametric clustering to find groups of hosts with similar communication patterns. By correlating this information with data on botnet activities (e.g., sending SPAM, scanning, etc...), they were able to improve botnet detection accuracy. They also address some potential evasion techniques that botnets can use to evade their detection techniques including real traffic mimicry and covert channels.

Zeng et al. expand on the work of BotMiner [66] to correlate network traffic analysis with host-based data collection. This allows greater flexibility for deployment because it alleviates the privacy issues associated with deep packet inspection, and it provides higher quality data to the detector because anonymous network traffic analysis can be error-prone [67].

Traffic analysis and intrusion detection experimentation often requires real labeled network training data. Several efforts have attempted to provide better labeling. Trestian et al. develop heuristics to classify end host application usage in a target network by searching for freely available or unintentionally leaked information from logs, forums, lists, and peer-to-peer tracking [68]. This technique is especially powerful when little to no network trace data is available. Gringoli et al. developed the gt architecture for collecting host-level information to establish the ground truth about the protocol and even application origins of network packet data. In future work, we could use gt to collect application information to better label the training data we used with our classification attack in Section 3.1.

6.2 Preventing Traffic Analysis

One commonly studied method for preventing traffic analysis is to add packets to the stream to remove the correlation between the plaintext protocol and its encrypted equivalent. Guan et al. developed NetCamo, which provides traffic analysis resistance by adding and rerouting packets with strict quality of service bounds [69]. In later work, they investigated how constant and variable packet inter-arrival times resist traffic analysis [70]. Wang et al. propose methods to optimally schedule real and padding packets in low-latency anonymity systems to prevent flow linking attacks [71]. Levine et al. investigated timing information leaked by low-latency mix systems [14]. They propose a novel traffic analysis prevention technique called *defensive dropping* to be used in concert with constant-rate padding. Since any legitimate packet drops in a constant-rate stream of packets will increase its correlation through a mix, they propose injecting such drop events randomly into the stream.

Another defense mechanism is to pad data at the application rather than packet layer. Liberatore and Levine evaluate several methods for determining the identities of pages in an encrypted Web browsing session [6]. To defend against their attack, they propose several effective, yet high overhead, application layer link padding methods. Other systems use cover traffic mixed with peer traffic and padding (e.g., Tarzan [72]) or plaintext blending (e.g., Infranet [73]) to evade traffic analysis.

Similar to our work, Wright et al. propose mapping real traffic patterns onto an existing real cover traffic model [74]. They use a linear optimization to find a solution to map packet sizes from the real traffic to the target packet size distribution. They evaluate their technique with success against an HTTP Web site identification attack as well as VoIP phrase detection. Differing from our work, their work focuses solely on packet sizes, rather than timings, and assumes that the former are either independently and identically distributed or from short packet sequences. They do not address the degree to which their technique prevents defense detection, especially when looking at longer packet sequences. Gianvecchio et al. developed a covert timing channel based on learning a distribution of inter-packet delays from specified input traffic [75]. They likewise assume that inter-packet delays are i.i.d. Our work produces greater realism since it mimics protocol features that are more than one or two packets long.

6.3 Traffic Modeling and Generation

Traffic generation has been the subject of study in the simulation, modeling, and performance assessment fields for some time. Being a dominant protocol on the Internet for some time, previous work has focused heavily on modeling HTTP traffic. Mah performed one of the first measurement studies of empirical Web packet trace data [76]. He found that most features did not have a good enough fit to a known distribution, so he used inverse transform sampling of empirical distributions to model traffic (as we do for TrafficMimic).

Barford and Crovella also developed a model for HTTP traffic called SURGE [39]. Differing from the work of Mah, SURGE uses several structural components and fits the empirical data to known parametrized distributions. Harpoon later automated this process for generic protocols using a network trace to configure the generator [40]. It collects empirical statistics from an input trace on file sizes, interconnection times, IP addresses, and the number of active sessions and then uses this empirical data to play back UDP transfers.

Cao et al. developed a connection-based model of HTTP traffic [77]. They argue that the connection/network model better captures the behavior of HTTP *on the network* than the page-oriented models often used to test Web server performance. In a similar vein, Swing [15] generates realistic network traffic for use in an emulation environment using generic network traffic models. It uses empirical distributions taken from a network trace of users, applications, connections, and the network to generate realistic TCP traffic.

Kannan et al. focus on modeling and reconstructing user sessions which span different connections and even protocols [78]. They represent user sessions using deterministic finite automata, which capture the order, type, and directionality of the connections composing a session. In future work, TrafficMimic could incorporate cross connection session features in the generation of cover traffic.

CHAPTER 7

CONCLUSION

Of the myriad challenges to communicating securely over a network, preventing traffic analysis is perhaps the most daunting. Seemingly insignificant information leakage can compromise the security of encryption, key management, and secure protocol design. The growing use of encryption is forcing traffic analysis into the mainstream of privacy attacks, while effective and practical defenses have seen only minimal deployment. Through our work on this dissertation, we have begun to address this shortcoming. We have presented the design, implementation, and evaluation of a realistic cover traffic tunneling system called TrafficMimic, which strongly resists traffic analysis and defense detection.

Through our use of realistic cover traffic, TrafficMimic addresses both traditional traffic analysis as well as defense detection attacks. We have shown that traffic classification, anomaly detection, and defense detection attacks are all unable to differentiate real traffic from our synthetic cover traffic. Through our use of independent traffic models or mutual information bounded biasing, we ensure that minimal information about the real traffic is leaked to the attacker. Because we based the attacks on state-of-the-art techniques (including clustering, K-NN classification, Bayesian inference, KS-testing), allowed the attacker considerable resources and access (including large varied training sets where we assume port-based labeling is correct), and ensured that the attacks were practical for use on real networks (by validating our techniques using independent validation sets collected at different times and on different networks), we have high confidence that our defenses work. While these attacks do not represent the full spectrum of potential attacks against TrafficMimic, they are sufficiently powerful and varied to illustrate the robustness of TrafficMimic.

Since our aim is to address traffic analysis attacks in real systems, we devoted considerable effort to the practicality of the implementation and performance of TrafficMimic. We found independent cover traffic modeling was very secure but suffered poor performance especially for common Internet tasks (e.g., chat or Web browsing). After careful study of independent cover traffic tunneling with simulation and a bi-directional analytic model, we addressed the performance gap by developing cover traffic parameter biasing methods, which tune the cover tunnel to the requirements of the real traffic. Since this technique has the potential to leak information, we developed attacks and methods to understand the information leakage and the affects biasing have on defense detection. By using these performance enhancements, we were found a multiple fold increase in the performance of TrafficMimic tunneling.

These performance improvements provide users with a variety of choices of cover traffic models with differing properties for information leakage, defense detection resistance, bandwidth, and latency. Since standard encrypted protocols provide minimal resistance to traffic analysis, these choices widen the spectrum of options for a privacy seeking user. Through our work, we have created a range of traffic analysis defenses that allow users to balance performance with better security.

7.1 Future Work

There are several areas of future research that will improve the fidelity, performance, and security of TrafficMimic. First, we expect that learning models of traffic for secure traffic generation can be improved. For example, we can encapsulate more network traffic details like multi-connection or multi-port sessions. We can also capture additional security-sensitive details like careful mimicry of the plaintext portions of the SSL handshake and connection setup.

Second, we can further improve performance by biasing other cover traffic parameters. As discussed in Chapter 5, our biasing techniques are general enough to be applied to other parameters like inter-connection time, number of exchanges, etc.. These enhancements can provide better interactive latency for real-time sessions or chat. This would further expand the security and performance options offered by realistic cover traffic generation.

The third opportunity for future work is tighter integration of TrafficMimic into common privacy enhancing technologies. We have already shown how TrafficMimic can integrate in several ways to the Tor anonymity network. To help the adoption of TrafficMimic within Tor, we intend to perform a Tor-specific performance and security evaluation. Since Tor is capacity limited, we may need to tune our bias algorithms to minimize excess padding while still providing the SSL traffic analysis resistance that Tor currently lacks.

We also intend to investigate how TrafficMimic can implement a VPN, which tunnels all of a user's traffic through multiple cover traffic streams. We have already performed some analysis that suggests that packet-oriented tunneling has certain performance advantages. We will investigate this further both through additional simulation study and evaluating a real TrafficMimic VPN implementation.

Lastly, we also plan to investigate how TrafficMimic functionality can be encapsulated into a library for inclusion with arbitrary programs. We could implement this feature using a library wrapper around the SSL library similar to the way transparent socks proxies wrap the socket interface. This would broaden the adoption of TrafficMimic by allowing developers and users alike to seamlessly integrate traffic analysis into any application that already uses SSL encryption for its network communications.

REFERENCES

[1] R. Atkinson, "Security Architecture for the Internet Protocol," RFC 1825 (Proposed Standard), Aug. 1995, obsoleted by RFC 2401. [Online]. Available: http://www.ietf.org/rfc/rfc1825.txt

[2] D. J. Barrett, R. E. Silverman, and R. G. Byrnes, *SSH, the Secure Shell: The Definitive Guide*. O'Reilly Media, Inc., 2005.

[3] N. Modadugu and E. Rescorla, "The Design and Implementation of Datagram TLS," in *NDSS*, 2004.

[4] E. Rescorla, *SSL and TLS: Designing and Building Secure Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[5] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine, "Privacy Vulnerabilities in Encrypted HTTP Streams," in *Privacy Enhancing Technologies*, 2005, pp. 1–11.

[6] M. Liberatore and B. N. Levine, "Inferring the Source of Encrypted HTTP Connections," in *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2006, pp. 255–263.

[7] Q. Sun, D. Simon, Y.-M. Wang, W. Russell, V. Padmanabhan, and L. Qiu, "Statistical Identification of Encrypted Web Browsing Traffic," in *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, 2002, pp. 19–30.

[8] F. Monrose and A. Rubin, "Authentication via Keystroke Dynamics," *CCS '97: Proceedings of the 4th ACM conference on Computer and communications security*, pp. 48–56, 1997.

[9] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2001, pp. 25–25.

[10] 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 *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 35–49.

[11] C. V. Wright, L. Ballard, F. Monrose, and G. M. Masson, "Language Identification of Encrypted VoIP Traffic: Alejandra y Roberto or Alice and Bob?" in *Proceedings of the 16th Usenix Security Symposium*, 2007. [Online]. Available: \url{http://www.usenix.org/events/sec07/tech/wright.html}

[12] R. Dhamankar and R. King, "PISA: Protocol Identification via Statistical Analysis," Blackhat US, 2007.

[13] C. V. Wright, F. Monrose, and G. M. Masson, "On Inferring Application Protocol Behaviors in Encrypted Network Traffic," *Journal of Machine Learning Research*, vol. 6, pp. 2745–2769, 2006.

[14] B. N. Levine, M. K. Reiter, C. Wang, and M. K. Wright, "Timing Attacks in Low-Latency Mix-Based Systems," in *Proceedings of Financial Cryptography (FC '04)*, A. Juels, Ed. Springer-Verlag, LNCS 3110, February 2004, pp. 251–265.

[15] K. V. Vishwanath and A. Vahdat, "Realistic and Responsive Network Traffic Generation," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 111–122, 2006.

[16] R. Dingledine, N. Mathewson, and P. F. Syverson, "Tor: The Second-Generation Onion Router," in *USENIX Security Symposium*, 2004, pp. 303–320.

[17] A. W. Moore and D. Zuev, "Internet Traffic Classification using Bayesian Analysis Techniques," in *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2005, pp. 50–60.

[18] D. M. Nicol, J. Liu, and M. Liljenstam, "Simulation of Large-Scale Networks Using SSF," in *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, LA, December 2003, pp. 650–657.

[19] "WireShark: A Network Protocol Analyzer for Windows and Unix," http://www.wireshark.org.

[20] G. Danezis and R. Clayton, "Introducing Traffic Analysis," in *Digital Privacy: Theory, Technologies and Practices*, A. Acquisti, S. Gritzalis, C. Lambrinoudakis, and S. di Vimercati, Eds. Auerbach Publications (Taylor and Francis Group), Nov 2007.

[21] D. Wagner and B. Schneier, "Analysis of the SSL 3.0 Protocol," in *WOEC'96: Proceedings of the 2nd USENIX Workshop on Electronic Commerce*. Berkeley, CA, USA: USENIX Association, 1996, pp. 29–40.

[22] J.-F. Raymond, "Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems," in *International workshop on Designing privacy enhancing technologies*. New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 10–29.

[23] G. Danezis, “The Traffic Analysis of Continuous-Time Mixes,” *Privacy Enhancing Technologies*, pp. 35–50, 2004.

[24] L. Øverlier and P. Syverson, “Locating Hidden Servers,” in *IEEE Symposium on Security and Privacy*, 2006.

[25] A. Hintz, “Fingerprinting Websites Using Traffic Analysis,” *Privacy Enhancing Technologies*, pp. 171–178, 2002.

[26] Y. Zhu and R. Bettati, “Un-mixing Mix Traffic,” in *In Proc. of Privacy Enhancing Technologies workshop (PET 2005)*, 2005.

[27] W. Dai, “Two Attacks Against Freedom,” http://www.eskimo.com/weidai/freedom-attacks.txt, 2000.

[28] Y. Zhang and V. Paxson, “Detecting Stepping Stones,” in *SSYM’00: Proceedings of the 9th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2000, pp. 13–13.

[29] T. He and L. Tong, “Detecting Encrypted Stepping-Stone Connections,” *IEEE Transactions on Signal Processing*, vol. 55, no. 5, pp. 1612–1623, May 2007.

[30] X. Wang, D. Reeves, and S. F. Wu, “Inter-Packet Delay Based Correlation for Tracing Encrypted Connections Through Stepping Stones,” in *European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, D. Gollmann, G. Karjoth, and M. Waidner, Eds., vol. 2502. Springer, Oct. 2002, pp. 244–263.

[31] D. Donoho, A. Flesia, U. Shankar, V. Paxson, J. Coit, and S. Staniford, “Multiscale Stepping-stone Detection: Detecting Pairs of Jittered Interactive Streams by Exploiting Maximum Tolerable Delay,” in *International Symposium on Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, A. Wespi, G. Vigna, and L. Deri, Eds., vol. 2516. Springer, Oct. 2002, pp. 16–18.

[32] A. Blum, D. X. Song, and S. Venkataraman, “Detection of Interactive Stepping Stones: Algorithms and Confidence Bounds,” in *International Symposium on Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, E. Jonsson, A. Valdes, and M. Almgren, Eds., vol. 3224. Springer, Sep. 2004, pp. 258–277.

[33] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer, “Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection,” in *Proceedings of the 15th Usenix Security Symposium*, Vancouver, B.C, Canada, 2006, pp. 257–272.

[34] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker, “Unexpected Means of Protocol Inference,” in *IMC ’06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM, 2006, pp. 313–326.

[35] C. Wright, F. Monrose, and G. M. Masson, "HMM Profiles for Network Traffic Classification," *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and Data Mining for Computer Security*, pp. 9–15, 2004.

[36] N. Xu, S.-Q. Chen, and H.-F. Sui, "Design and Implementation of HTTPS Traffic Capturing and Parsing System," *Journal of Central South University (Science and Technology)*, vol. 36, no. 4, pp. 664–667, August 2005.

[37] H. Cheng and R. Avnur, "Traffic Analysis of SSL Encrypted Web Browsing," http://www.cs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-reports/ronathan-heyning.ps.

[38] J.-Q. Shi, B.-X. Fang, B. Li, and F.-L. Wang, "Using Support Vector Machine in Traffic Analysis for Website Recognition," in *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, vol. 5, August 2004, pp. 2680– 2684.

[39] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 151–160, 1998.

[40] J. Sommers and P. Barford, "Self-Configuring Network Traffic Generation," in *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM, 2004, pp. 68–81.

[41] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee, "Polymorphic Blending Attacks," in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006.

[42] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. I. Sharif, "Misleading Worm Signature Generators Using Deliberate Noise Injection," in *IEEE Symposium on Security and Privacy*, 2006, pp. 17–31.

[43] W. Willinger, V. Paxson, and M. S. Taqqu, "Self-Similarity and Heavy Tails: Structural Modeling of Network Traffic," in *A Practical Guide to Heavy Tails: Statistical Techniques and Applications*. Cambridge, MA, USA: Birkhauser Boston Inc., 1998, pp. 27–53.

[44] OpenBSD, "Openssh manual pages," http://www.openssh.org/manual.html.

[45] Google, "Protocol buffers," http://code.google.com/p/protobuf/.

[46] D. Herrmann, R. Wendolsky, and H. Federrath, "Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier," in *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security*. New York, NY, USA: ACM, 2009, pp. 31–42.

[47] R. Dingledine and N. Mathewson, "Design of a Blocking-Resistant Anonymity System," https://svn.torproject.org/svn/projects/design-paper/blocking.pdf, December 2006.

[48] C. Walsworth, E. Aben, kc claffy, and D. Andersen, "The CAIDA Anonymized 2009 Internet Traces," http://www.caida.org/data/passive/passive_2009_dataset.xml, collected Jan 15, and Feb 19, 2009 at equinix-chicago monitor.

[49] UNC Distributed and Real-Time Systems Group, "UNC April/May 2003 Packet Header Traces (collection)," http://imdc.datcat.org/collection/1-0299-L=UNC+April\%2FMay+2003+packet+header+traces.

[50] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak, "Operating System Support for Planetary-Scale Network Services," in *NSDI'04: Proceedings of the 1st Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 253–266.

[51] R. Wendolsky, D. Herrmann, and H. Federrath, "Performance Comparison of Low-Latency Anonymisation Services from a User Perspective," in *Privacy Enhancing Technologies*, 2007, pp. 233–253.

[52] N. Kiyavash, A. Houmansadr, and N. Borisov, "Multi-flow Attacks Against Network Flow Watermarking Schemes," in *SS'08: Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 307–320.

[53] M. Gates and A. Warshavsky, "Iperf," http://iperf.sourceforge.net.

[54] D. M. Nicol and N. Schear, "Models of Privacy Preserving Traffic Tunneling," *Simulation*, vol. 85, no. 9, pp. 589–607, 2009.

[55] N. Schear and D. M. Nicol, "Performance Analysis of Real Traffic Carried with Encrypted Cover Flows," in *Proceedings of the 2008 Conference on Principles of Advanced and Distributed Simulation*, Rome, Italy, June 2008, pp. 80–87.

[56] S. Ross, *Stochastic Processes, 2nd Edition*. New York: Wiley, 1996.

[57] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*. London: Chapman and Hall, 1986.

[58] R. A. Ince, R. S. Petersen, D. C. Swan, and S. Panzeri, "Python for Information Theoretic Analysis of Neural Data," *Frontiers in Neuroinformatics*, vol. 4, no. 0, p. 12, 2010.

[59] M. A. Stephens, "Use of the Kolmogorov-Smirnov, Cramer-Von Mises and Related Statistics Without Extensive Tables," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 32, no. 1, pp. pp. 115–122, 1970.

[60] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C (2nd ed.): The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 1992.

[61] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "BLINC: Multilevel Traffic Classification in the Dark," *SIGCOMM Comput. Commun. Rev.*, vol. 35, pp. 229–240, August 2005. [Online]. Available: http://doi.acm.org/10.1145/1090191.1080119

[62] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese, "Network Monitoring using Traffic Dispersion Graphs (TDGs)," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1298306.1298349 pp. 315–320.

[63] M. Iliofotou, M. Faloutsos, and M. Mitzenmacher, "Exploiting Dynamicity in Graph-based Traffic Analysis: Techniques and Applications," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, ser. CoNEXT '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1658939.1658967 pp. 241–252.

[64] G. Gu, J. Zhang, and W. Lee, "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[65] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-independent Botnet Detection," in *SS'08: Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 139–154.

[66] Y. Zeng, X. Hu, and K. Shin, "Detection of Botnets using Combined Host- and Network-level Information," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, 2010, pp. 291 –300.

[67] S. E. Coull, M. P. Collins, C. V. Wright, F. Monrose, and M. K. Reiter, "On Web Browsing Privacy in Anonymized NetFlows," in *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–14.

[68] I. Trestian, S. Ranjan, A. Kuzmanovi, and A. Nucci, "Unconstrained Endpoint Profiling (Googling the Internet)," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 279–290, August 2008. [Online]. Available: http://doi.acm.org/10.1145/1402946.1402991

[69] Y. Guan, X. Fu, D. Xuan, P. Shenoy, R. Bettati, and W. Zhao, "NetCamo: Camouflaging Network Traffic for QoS-Guaranteed Mission Critical Applications," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 31, no. 4, pp. 253–265, Jul 2001.

[70] X. Fu, B. Graham, R. Bettati, W. Zhao, and D. Xuan, “Analytical and Empirical Analysis of Countermeasures to Traffic Analysis Attacks,” in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, Oct. 2003, pp. 483–492.

[71] W. Wang, M. Motani, and V. Srinivasan, “Dependent Link Padding Algorithms for Low Latency Anonymity Systems,” in *ACM Conference on Computer and Communications Security*, 2008, pp. 323–332.

[72] M. J. Freedman and R. Morris, “Tarzan: A Peer-to-peer Anonymizing Network Layer,” in *CCS ’02: Proceedings of the 9th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2002, pp. 193–206.

[73] N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan, and D. Karger, “Infranet: Circumventing web censorship and surveillance,” in *Proceedings of the 11th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2002, pp. 247–262.

[74] C. Wright, S. Coull, and F. Monrose, “Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis,” in *NDSS 09: Proceedings of the 14th Annual Network and Distributed Systems Symposium*, Feb 2009.

[75] S. Gianvecchio, H. Wang, D. Wijesekera, and S. Jajodia, “Model-Based Covert Timing Channels: Automated Modeling and Evasion,” in *RAID ’08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 211–230.

[76] B. Mah, “An Empirical Model of HTTP Network Traffic,” in *INFOCOM ’97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 2, Apr. 1997, pp. 592 –600 vol.2.

[77] J. Cao, W. Cleveland, Y. Gao, K. Jeffay, F. Smith, and M. Weigle, “Stochastic Models for Generating Synthetic HTTP Source Traffic,” in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 3, 2004, pp. 1546 –1557 vol.3.

[78] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal, “Semi-automated Discovery of Application Session Structure,” in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, ser. IMC ’06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1177080.1177096 pp. 119–132.

AUTHOR'S BIOGRAPHY

Nabil Schear received a B.S. in Computer Science with highest honors and cooperative program distinctions from the Georgia Institute of Technology in 2004. He received a M.S. degree in Computer Science from the University of California at San Diego in 2007 under the advisement of Amin Vahdat. He will complete his Ph.D. in 2011 at the University of Illinois at Urbana-Champaign Department of Computer Science under the advisement of Nikita Borisov. He worked as intern at Los Alamos National Laboratory starting 2001 and joined the technical staff there in 2004 where he remains. His research interests include vulnerability assessment, secure communications, and intrusion detection.