Performance of Selected Noisy Covert Channels and Their Countermeasures in IP Networks

Thesis submitted in accordance with the requirements
for the degree of Doctor of Philosophy

Sebastian Zander

Centre for Advanced Internet Architectures
Faculty of Information and Communication Technologies
Swinburne University of Technology
Melbourne

Abstract

Encryption alone secures communication by preventing adversaries from easily decoding one's transmissions. Covert channels go one step further by attempting to hide the very existence of communication. They hide inside legitimate overt network traffic. Huge amounts of traffic make the Internet an ideal vehicle for covert communications.

Most existing covert channels are simple and in principle easy to detect or eliminate. The more complex channels are usually harder to detect and eliminate, but typically suffer from channel noise. Previous work has only partly analysed the performance of noisy channels and their countermeasures and has not compared different types of channels.

We characterise the trade-offs between channel simplicity, capacity and ease of detection and elimination by investigating the performance of selected noisy covert channels and their countermeasures. Not all chosen channels are entirely new, but we propose novel improved encoding schemes. We also develop techniques for reliable data transmission. We analyse the theoretical channel capacities as well as empirically measure achievable throughputs. We show that the Internet's potential to support more sophisticated covert channels is considerably greater than suggested by most existing simple channels.

First, we analyse a channel in the IP Time-to-live (TTL) header field. We develop new stealthier encoding schemes that also provide a slightly increased capacity. The channel has a comparatively high capacity of up to a few hundred bits per second depending on the overt traffic, but is easy to detect and eliminate. Next, we analyse an inter-packet gap timing channel. We develop novel stealthy encoding schemes because previous schemes are easy to detect. The channel only has up to 70–80% of the TTL channel's capacity, but is harder to detect. However, it can still be eliminated.

Then we propose and analyse a novel indirect channel in multiplayer game traffic. The channel is impractical to eliminate, but is still detectable. The capacity is up to 10–20 bits per second – lower than that of direct channels. Next, we analyse an indirect timing channel that transmits bits via temperature changes. We develop an improved version of the channel that has increased capacity. Still the capacity is only 10–20 bits per hour, but the channel is potentially hard to detect and eliminate.

Finally, we develop techniques to detect and eliminate the covert channels and evaluate their effectiveness. While the proposed elimination methods are effective but channel-specific, we demonstrate that machine-learning techniques detect different covert channels with over 95% accuracy.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my supervisors, Grenville Armitage and Philip Branch, for their excellent advice and constant support throughout my time at Swinburne University.

I thank all members of the Centre for Advanced Internet Architectures (CAIA), both past and present, for our interesting and fruitful discussions. Many thanks to Lam Hoang Do, Warren Harrop, Imrul Hassan, Amiel Hyde, Carl Javier, Tung Le, Lucas Parry, Mattia Rossi, Lawrence Stewart and Kewin Stoeckigt for helping me to collect Quake III traffic data. Thanks to David Hayes for the LaTeX tips and keeping my testbed up and running when I was working from home.

Special thanks to Amiel Hyde for giving me access to the Skype traffic traces, Richard Nelson for giving me access to the Waikato traffic trace and Chris Meyers for making it possible to capture the Grangenet traffic.

I am grateful for the comments from my anonymous examiners. However, the responsibility for all the remaining errors and omissions rests with me.

I am thankful to Swinburne University for supporting me with a Chancellors Research Scholarship. I thank Swinburne library, in particular Meredith Wilkinson, for helping me to retrieve difficult-to-get literature.

Furthermore, I would like to thank Markus Kuhn, Steven Murdoch and Andrew Moore from Cambridge University, where I spend three months as visiting student, for their great hospitality and advice.

I am indebted to my family, especially my wife Wunna and my parents Karin and Horst, for their support and advice. I also thank my friends for making my life interesting and enjoyable.

Many thanks to a number of people for answering my questions or giving me access to their software: Jun Hu and Tolga Duman, Steven Gianvecchio and Roy Yates.

Last but not least I thank the many people who created the following magnificent open source software packages without I could not have produced this thesis: LaTeX, LyX, Miktex, R, and WEKA.

Dedicated to Wunna and Lukas for their love and support.

Declaration

To the best of my knowledge and belief, this thesis contains no material previously published or written by any other person, except where due reference is made in the text of the thesis. This thesis has not been submitted previously, in whole or in part, to qualify for any other degree or diploma. The content of the thesis is the result of work, which has been carried out since the beginning of my candidature in May 2006. Where the work is based on joint research or publications, the thesis discloses the relative contributions of the respective workers or authors.

Melbourne, 4th of May 2010

Sebastian Zander

Published Work and Collaboration

During this thesis I have published the following conference papers and journal articles:

[1] S. Zander, G. Armitage, P. Branch. *Reliable Transmission Over Covert Channels in First Person Shooter Multiplayer Games*. In Proceedings of 34th Annual IEEE Conference on Local Computer Networks (LCN), October 2009.

[2] S. Zander, G. Armitage, P. Branch. *Covert Channels in Multiplayer First Person Shooter Online Games*. In Proceedings of 33rd Annual IEEE Conference on Local Computer Networks (LCN), October 2008.

[3] S. Zander, S. J. Murdoch. *An Improved Clock-skew Measurement Technique for Revealing Hidden Services*. In Proceedings of 17th Usenix Security Symposium, July/August 2008.

[4] S. Zander, G. Armitage, P. Branch. Covert Channels and Countermeasures in Computer Network Protocols. (invited) *IEEE Communications Magazine*, vol. 45, pp. 136–142, December 2007.

[5] S. Zander, G. Armitage, P. Branch. *An Empirical Evaluation of IP Time To Live Covert Channels*. In Proceedings of IEEE International Conference on Networks (ICON), November 2007.

[6] S. Zander, G. Armitage, P. Branch. A Survey of Covert Channels and Countermeasures in Computer Network Protocols. *IEEE Communications Surveys and Tutorials (COMST)*, 9(3):44–57, October 2007.

[7] S. Zander, P. Branch, G. Armitage. *Error Probability Analysis of IP Time To Live Covert Channels*. In Proceedings of 7th IEEE International Symposium on Communications and Information Technologies (ISCIT), October 2007.

[8] S. Zander, G. Armitage, P. Branch. *Covert Channels in the IP Time To Live Field*. In Proceedings of Australian Telecommunication Networks and Applications Conference (ATNAC), December 2006.

I also have co-authored a number of CAIA tech reports [9, 10, 11, 12].

Chapter 2 is based on the IEEE COMST article [6], but has been restructured and updated with recent work. The IEEE COMST editorial board selected this "outstanding article" to be reprinted and featured in IEEE Communications Magazine [4].

Papers [8], [7] and [5] analyse the performance of the TTL covert channel. They form the basis of Chapter 3. However, the chapter significantly expands the research beyond the content of these papers and a larger part of the work has not yet been published.

The work in Chapter 4 was done only recently and has not yet been published. A paper has been submitted for publication.

Paper [2] develops and analyses a novel covert channel in multiplayer game traffic. Paper [1] develops and evaluates an efficient mechanism for reliable data transport over this unreliable channel. Both papers form the basis of Chapter 5. However, the chapter contains further discussion and additional results not part of the published papers.

The first part of Chapter 6 that develops and evaluates an improved version of the covert channel is based on [3]. The second part of the chapter, the analysis of the channel capacity, has been submitted for publication.

Paper [3] was written in collaboration with Steven Murdoch. A large part of the work was carried out during my three-month visit of the Computer Laboratory of Cambridge University, UK. Steven proposed the initial idea for the improved remote clock-skew estimation algorithm and we developed the new attack scenarios together. I designed, implemented and evaluated the actual algorithm.

CONTENTS

Appendices

List of Abbreviations

ACF	Auto Correlation Function
ADSL	Asymmetric Digital Subscriber Line
AH	Authentication Header
AODV	Ad-hoc On-Demand Distance Vector
ARQ	Automatic Repeat Request
ATM	Asynchronous Transfer Mode
AWGN	Additive White Gaussian Noise
BSC	Binary Symmetric Channel
BAC	Binary Asymmetric Channel
CCHEF	Covert Channels Evaluation Framework
CC	Common Criteria
CDF	Cumulative Density Function
CFT	Covert Flow Tree
CPU	Central Processing Unit
CRC	Cyclic Redundancy Checksum
CSMA/CD	Carrier Sense Multiple Access Collision Detection
CSPRNG	Cryptographically Secure Pseudo Random Number Generator
CTS	Clear To Send
DF	Don’t Fragment
DNS	Domain Name System
DSL	Digital Subscriber Line
DoD	Department of Defense
DoS	Denial of Service
EPL	Evaluated Products List
ERF	Endace Record Format
ESP	Encapsulating Security Payload
FCFS	First Come First Serve
FEC	Forward Error Correction
FPS	First Person Shooter
FPSCC	First Person Shooter Covert Channel
FTP	File Transfer Protocol
HDLC	High-Level Data Link Control
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPG	Inter Packet Gap
IQR	Inter-Quartile Range
ISP	Internet Service Provider
ISN	Initial Sequence Number
ITSEC	Information Technology Security Evaluation Criteria

KS	Kolmogorov-Smirnov
LAN	Local Area Network
LDPC	Low-Density Parity Check
LSB	Least Significant Bit
MAC	Message Authentication Code
MAC	Medium Access Control
ML	Machine Learning
MLS	Multi Level Secure
MTU	Maximum Transfer Unit
NAT	Network Address Translation/Translator
NIC	Network Interface Card
NTP	Network Time Protocol
OSI	Open Systems Interconnection
PID	Proportional Integral Derivative
PLL	Phase Lock Loop
PPM	Parts Per Million
Q3	Quake III Arena
QQ	Quantile Quantile
RFPSCC	Reliable FPSCC
RMSE	Root Mean Square Error
ROC	Receiver Operating Characteristics
RS	Reed-Solomon
RTCP	Real-time Control Protocol
RTP	Real-time Transport Protocol
RTS	Ready To Send
RTT	Round Trip Time
SAFP	Store And Forward Protocol
SLOC	Source Lines of Code
SNR	Signal to Noise Ratio
SOF	Start Of Frame
SRM	Shared Resource Matrix
SSH	Secure Shell
TCP	Transmission Control Protocol
TCSEC	Trusted Computer System Evaluation Criteria
TOS	Type of Service
TTL	Time To Live
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VoIP	Voice over IP
WEP	Wired Equivalent Privacy
XML	Extensible Markup Language

Chapter 1 Introduction

Often it is thought that encryption is sufficient to secure communication. However, encryption only prevents unauthorised parties from decoding the communication. In many cases the mere existence of communication or changes in communication patterns, such as an increased message frequency, are enough to raise suspicion and reveal the onset of events. Covert channels attempt to hide the very existence of communication. Typically, they use means of communication not normally intended to be used.

Lampson introduced covert channels in 1973 in the context of monolithic Multi Level Secure (MLS) systems running on mainframes as a mechanism enabling a process at a high security level to leak information to a process at a low security level by exploiting shared resources, such as CPU, memory, or mass storage [13]. For example, a file's lock status in the file descriptor table can be used as a covert channel [14].

Traditionally, mainframes ran multiple processes with different security levels, but today these processes typically run on different hosts connected by computer networks. Firewalls control the security policies, but covert channels in network protocols can be used to leak information from a high security host to a low security host. Overt channels, such as legitimate network protocols, are used as carriers for covert channels [15, 16].

These days the applications for covert channels have extended far beyond MLS systems. A diverse range of individuals and groups has found reason to utilise covert channels for communication, and there are also a few specific applications. The huge amount of data and large number of protocols make the Internet an ideal vehicle for covert communication. The capacity of covert channels has greatly increased in recent years because of new high-speed network technologies, and this trend is likely to continue.

Covert channels in network protocols are similar to steganography [17], the hiding of information in audio, visual or textual content and therefore sometimes also referred to as "network steganography". While steganography requires some form of content as cover, covert channels require some network protocol as carrier. The ubiquitous presence of network protocols suitable as carriers (e.g. the Internet Protocol) make covert channels widely available, even in situations where steganography cannot be used.

It is important to note that covert channels are covert by definition, because they use unintended means of communication. They are 'invisible' for an unknowing adversary but not necessarily undetectable. In fact many existing covert channels can actually be detected if they are known to the adversary. Even if particular channels are hard to detect often they can still be eliminated.

We do not assume covert channels to be ‘good’ or ‘bad’, since this mainly depends on one’s particular viewpoint. Nevertheless, many applications of covert channels are serious threats to network security. Security incidents are usually kept secret, but evidence suggests that covert channels have been used successfully [18]. While covert channels may not have been widely used yet, it is likely that they will become more popular because of increased security measures against ‘open channels’, such as the transfer of memory sticks in and out of organisations and more sophisticated censorship of network traffic.

Early network covert channels are typically simple to implement, but on the other hand are also easy to detect or eliminate. As security systems are becoming capable of detecting and eliminating simple channels, more complex and harder to counter channels are emerging, which typically suffer from channel noise. This makes them more difficult to analyse and is probably the reason why their performance has not been well studied until now. We aim to gain a better understanding of these important channels. The goal is not to analyse every possible channel, but by choosing several representative channels, better understand their capacity, and techniques for their detection and elimination.

We analyse the performance of selected noisy covert channels and their countermeasures. We focus on channels that potentially can be used for general-purpose communication. Not all of the channels are novel, but we develop new improved encoding schemes for known channels. We analyse the theoretical channel capacities as well as empirically measure the achievable throughput based on channel noise. We consider not only channel-specific noise but also noise caused by packet loss and reordering of the overt traffic. Furthermore, we propose and evaluate techniques for reliable data transmission.

Our results show that most channel’s capacities are in the order of at least a few tens of bits per second up to a few hundreds of bits per second depending on the overt traffic, even for complex channels with non-ideal encoding schemes. The achievable throughputs are sufficient for transmitting text messages or smaller documents. Based on the traditional classification these are high-rate covert channels, as their rates are significantly above the often accepted maximum rate of one bit per second [19].

On the other hand the most complex channel we analyse has a capacity of significantly less than one bit per second. The channel seems less relevant for general-purpose communication, but it is very useful for specific applications. Our work characterises the trade-offs between simpler channels with higher capacity that are easier to counter and more complex channels with lower capacity that are harder to counter.

Understanding covert channels is crucial for developing countermeasures. After implementing and analysing the covert channels we then develop techniques to detect and eliminate the channels and evaluate their effectiveness. The proposed elimination methods are effective but channel-specific. On the other hand we demonstrate that more general machine-learning techniques are successful in detecting different covert channels.

1.1 Research objectives and contributions

This thesis analyses and compares the performance of selected noisy covert channels and their countermeasures. We compare the covert channels using theoretical models and practical experiments in real networks with different conditions. To the best of our knowledge such a comparison of different types of channels has not been done previously.

Most existing covert channels are simple and in principle easy to detect or eliminate. These channels are well understood and their capacities can be determined easily, but they are likely to be managed by upcoming security systems (see Section 2.4). Hence we focus on more complex channels that are harder to detect and eliminate, but typically experience channel noise. This makes them more difficult to analyse and is probably the reason why their performance has not been well studied in existing literature.

Covert channels either manipulate data fields or the timing of packets or messages (storage vs. timing), and covert data is either exchanged directly between covert sender and receiver or via an intermediate node (direct vs. indirect). Indirect channels have the benefit that an adversary does not see a direct flow of information between covert sender and receiver. For our analysis we select four channels that represent all combinations of these two characteristics.

Most of our selected channels are passive channels that can be embedded in existing overt traffic exchanged by unwitting senders and receivers. Passive channels relieve the covert sender of the difficult task of mimicking legitimate traffic properly, which is necessary to prevent detection due to suspiciously looking overt traffic. However, the covert sender can always generate traffic to create active channels. We chose an active indirect timing covert channel, because it is hard to construct passive indirect timing channels.

Furthermore, we only consider channels on or above the network layer, as they usually have a wider range and higher coverage than channels in specific link-layer protocols. Finally, we limit our choice to covert channels that impose no obvious adverse effects on the overt traffic (for example, we ignore channels that encode information by dropping or reordering overt packets). In particular we choose the following channels:

- a direct noisy storage channel in the IP Time-To-Live (TTL) header field,
- a direct noisy timing channel in IP inter-packet times,
- an indirect noisy storage channel in multiplayer game traffic, and
- an indirect noisy timing channel based on temperature changes.

The idea for three of the channels was previously published (see Section 2.2). However, flaws in existing encoding schemes led us to develop new improved encoding schemes in all cases. The covert channel in multiplayer game traffic is a novel type of covert channel.

Noisy covert channels usually not only experience bit errors but also synchronisation errors. Achieving rates near full capacity over such channels is challenging. Despite knowledge of the existence of such channels, they have not been well studied over the last decades [20]. Some coding schemes exist but often the error characteristics assumed in the literature do not match those of covert channels and the resulting performance remains unclear. Previous work usually focused only on the basic modulation mechanisms, but we also develop and evaluate techniques for reliable data transmission.

Developing and analysing the covert channels in the first step allows us to then develop detection and elimination techniques and evaluate their effectiveness in the following step. We explore the use of Machine Learning (ML) techniques for detecting the covert channels. The use of ML to detect network covert channels was proposed previously by a few researchers, but it remains a largely unexplored area.

Our approach to analysing and comparing the covert channels and countermeasures is based on empirical experiments across real networks and on experiments using traffic from trace files captured previously in real networks. However, we also use the concepts of information theory [21, 22] to estimate the capacity of the channels and then compare actual throughputs with estimated capacities. We choose to pursue a more experimental approach, since with pure theoretical analysis there remains a risk that research will ultimately be inapplicable to real systems.

This thesis contributes the following work, filling several gaps in existing literature:

1. We explore the trade-offs between channel simplicity, capacity and ease of detection and elimination by investigating the performance of selected noisy covert channels and their countermeasures. We propose a novel taxonomy that assisted us in our selection of the channels to investigate and helps to understand general characteristics of channels in each classification.

2. We explore direct noisy storage channels using the IP TTL field as a covert channel. We develop new encoding schemes that are both stealthier and have slightly larger capacity. We characterise the channel noise based on traffic traces. We propose a channel model that covers the effects of TTL noise, overt packet loss and reordering, and estimate the capacity. The channel has a comparatively high capacity of up to a few hundred bits per second, depending on the overt traffic's packet rate.

3. We investigate direct noisy timing channels based on covert channels in inter-packet times. We develop a novel variant of the channel and show that it is much harder to detect than previous versions. We propose a channel model and evaluate the

capacity. The channel has only 70–80% of the TTL channel's capacity. We also show that artificial network jitter almost completely eliminates the channel with only minor impact on the overt traffic's performance.

4. We develop a novel mechanism for reliable data transport over direct noisy covert channels. We demonstrate that it can be applied to TTL channels and inter-packet timing channels. The theoretical capacity is not reached, but our technique provides reasonable performance with throughputs of at least 30–40% of the capacity.

5. We explore indirect noisy storage channels based on a novel covert channel in First Person Shooter (FPS) multiplayer game traffic. Key advantages of the channel for users are that it is an indirect channel that cannot be eliminated without eliminating the game traffic. However, the capacity is only up to 10–20 bits per second. We develop a tailored scheme for reliable transport, measure the throughput depending on network conditions, and compare it with the theoretical capacity.

6. We analyse an indirect noisy timing covert channel that transmits information via changes of temperature. We first develop an improved version of the channel that increases the capacity, and evaluate its effectiveness. Then we develop a method to estimate the capacity depending on the characteristics of the intermediate host. For two example intermediate hosts the capacity is only 10–20 bits per hour.

7. We develop measures to detect and eliminate different noisy covert channels. The proposed elimination methods are effective but are usually channel-specific. We show that ML techniques are successful in detecting different covert channels with over 95% accuracy.

8. We develop an extensible software framework for creating and evaluating the different channels called Covert Channels Evaluation Framework (CCHEF). It can be used to create covert channels across real networks as well as emulate covert channels using overt traffic from trace files.

1.2 Significance of work

The handling of covert channels, such as their elimination and detection, is of particular importance for MLS systems. It is required for higher assurance levels in the US Department of Defense Trusted Computer System Evaluation Criteria (TCSEC) [19], the European Information Technology Security Evaluation Criteria (ITSEC) [23] and the Common Criteria (CC) [24]. The Evaluated Products List (EPL) [25] provides the Australian government with information about the compliance of products to the ITSEC and CC criteria.

Besides the area of MLS systems a wide field of possible applications for covert channels has opened with the rapid growth of the Internet (see Section 2.1.4). For example, it has been hypothesised that terrorists use covert channels for communication and coordination [26]. On the other hand they can be used by people to disseminate information secretly if there are serious repercussions against the "freedom of speech" [27].

In most cases the sensitivity of the subject prevents incidents from becoming publicly known. Because of this it was unclear for some time if covert channels actually had been exploited in the real world at all, but "at a workshop [in 1999], Bob Morris gave this question its final and complete answer: Yes." [18]. Network covert channels may not have been widely used yet, but it is likely that they will become more popular in the future because of increased security measures against 'open' channels.

Many existing covert channels are simple to detect and eliminate and thus will likely be handled by current or next-generation security systems. We analyse the next generation of more sophisticated covert channels under realistic network conditions. Our results show that even while these channels are noisy many of them still have sufficient capacity to be useful. Furthermore, we demonstrate that reasonably efficient techniques for reliable communication across such channels can be developed.

Our results highlight the trade-off between different covert channels. Channels with higher capacity are generally easier to counter whereas stealthier and more robust channels provide less capacity. While some covert channels can be eliminated but are hard to detect, others can be detected but are hard to eliminate. This means a holistic approach for covert channel handling is required to secure future networks. Our results suggest that ML techniques are a very promising approach to detect a wider range of covert channels.

The covert channel hidden inside multiplayer game traffic is an entirely new class of channels, usable for collusion as well as exchanging game-unrelated information unbeknownst to adversaries. Many companies are exploring the use of immersive virtual worlds similar to games, such as Second Life [28], for distributed training, collaboration and general business – this opens up the potential for covert ex-filtration of commercially sensitive information via such channels.

1.3 Thesis outline

We begin with an overview of existing covert channel techniques and countermeasures in Chapter 2. Next, Chapter 3 proposes new encoding schemes and analyses the performance of covert channels in the IP TTL header field. In Chapter 4 we develop new stealthier covert channels in the inter-packet times of IP packets and analyse their performance. In Chapter 5 we present a novel covert channel in FPS multiplayer game protocols and analyse its performance. In Chapter 6 we develop a technique to improve the capacity of temperature-based covert channels and estimate the capacity in example scenarios. In Chapter 7 we develop and analyse mechanisms to eliminate and detect the different covert channels. In Chapter 8 we present our conclusions and outline future work.

Chapter 2
Covert Channels

We begin the chapter with an overview of covert channels, and then discuss previous work on covert channels and countermeasures. We group existing covert channels according to a taxonomy we have developed. We conclude the chapter by identifying gaps in the literature that are addressed in this thesis.

2.1 Background

We first define important terminology and then explain the basic model for covert channels as well as possible communication scenarios. We also discuss various applications of covert channels and explain the generally available countermeasures. Finally, we discuss criteria for evaluating covert channels.

2.1.1 Terminology

Researchers have used a range of terms – such as covert channels, network steganography or information hiding – to describe the process of hiding information in network protocols. Partly this has been caused by differences between Lampson's original covert channel definition and a later definition by the US Department of Defense (DoD). Another reason is that terminology evolved: the term "information hiding" simply had not been coined when the first covert channels in network protocols were proposed [18].

Lampson defined covert channels as "channels, [...] not intended for information transfer at all" [13] whereas the US DoD TCSEC, commonly known as "Orange Book", defined covert channels as "[...] any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy" [19]. Newer definitions are broadly consistent with the Orange Book.

The Common Criteria defined a covert channel as "an enforced, illicit signalling channel that allows a user to surreptitiously contravene the multi-level separation policy and unobservability requirements of the [target of evaluation]" [24]. The "Light-Pink Book" used a more formal definition based on Tsai *et al.* [29]: "Given a non-discretionary (e.g., mandatory) security policy model M and its interpretation $I(M)$ in an operating system, any potential communication between two subjects $I(S_i)$ and $I(S_k)$ of $I(M)$ is covert if and only if any communication between the corresponding subjects S_i and S_k of the model M is illegal in M." [14].

We use the term *covert channel* when we refer to the hiding of information in network protocols and refer to the information transmitted across the channel as *hidden* or *covert information*. Consistently with Petitcolas *et al.* we use the term *steganography* (literally meaning covered writing) when we refer to the hiding of information in content, and the term *information hiding* as a generic term for both [17]. Transmission of information through legitimate network traffic is an *overt channel* [30]. Also, we refer to overt traffic containing an embedded covert channel as *cover traffic*.

The definition of a covert channel implies that both the sender and receiver collude to exchange information. A *side channel* is a covert channel where the sender unintentionally leaks information and only the receiver wants a successful communication. A *subliminal channel* is a covert channel inside a crypto system [31].

2.1.2 Prisoner problem

The prisoner problem is the de-facto model for covert channel communication [31]. Two people, Alice and Bob[1], are thrown into prison and intend to escape. To agree on an escape plan they need to communicate but Wendy the warden monitors all their messages. If Wendy finds any signs of suspicious messages she will place Alice and Bob into solitary confinement – making it impossible for them to escape. Alice and Bob must exchange innocuous messages containing hidden information that hopefully Wendy will not notice. Craver describes the different types of wardens [32]:

- A *passive* warden can only spy on the channel but cannot alter any messages.
- An *active* warden can modify messages slightly, but without altering the semantics.
- A *malicious* warden may alter the messages with impunity, but in reality malicious wardens are rare [32].

Handel *et al.* extended this scenario towards computer networks, where Alice and Bob use two networked computers for communication [16]. They run an innocuous looking overt communication channel between their computers, containing a hidden covert channel. Alice and Bob share a secret, used for determining covert channel encoding parameters and encrypting/authenticating the hidden messages. For practical purposes Alice and Bob may well be the same person, for example a hacker ex-filtrating restricted information. Wendy manages the network and monitors the passing traffic for covert channels or alters

[1]Cryptographic protocols are usually illustrated using participants named alphabetically (Alice, Bob) or with names where the first letter matches their role (Wendy the warden).

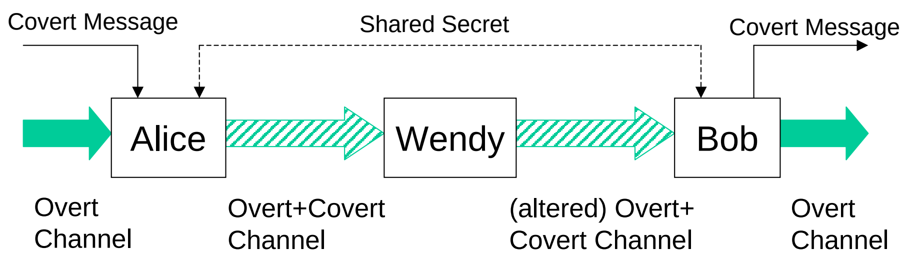


Figure 2.1: The prisoner problem – de-facto model for covert channel communication

the passing traffic to eliminate or disrupt covert channels. Figure 2.1 depicts the model with Alice sending to Bob.

In the prisoner model Alice communicates with Bob, but in general covert channels are not restricted to unicast channels. Alice could send hidden information to Bob, Carol and Dave at the same time if the channel allows multicast communication.

2.1.3 Communication scenarios

There are a number of different scenarios for covert communication depending on whether Alice and Bob are the sender and receiver of the overt channel, or if they act as *middlemen* and manipulate an overt channel between innocent users [33].

If Alice is also the sender of the overt channel, she can manipulate the overt channel as desired (e.g. to maximise capacity or stealth). However, sometimes Alice may not be able to create overt channels or may choose not to do so for increased stealth. In this case Alice can act as middleman embedding a covert channel into an existing overt channel. Obviously, then the capacity of the covert channel depends on the existing overt channel.

Bob can be the receiver of the overt channel, but to increase stealth he can also be a middleman extracting the hidden information from an overt communication destined for an innocent receiver. Then Bob should (if possible) remove the covert channel preventing possible detection by the receiver or any other upstream intermediate nodes.

Being a middleman does not necessarily mean Alice or Bob have to be physically separated from the overt sender and receiver. They could be located on routers or gateways between the overt sender and receiver, but they could also be on the same physical device located in lower levels of the network protocol stack.

Figure 2.2 illustrates the possible combinations of covert sender and receiver locations. The actual communication scenario also depends on the application of the covert channel. For example, if the channel is used to circumvent censorship covert and overt sender/receiver would likely be identical, whereas if it is used by a hacker for ex-filtrating data the covert sender and receiver would likely be middlemen (e.g. Alice could be inside the network protocol stack of the compromised host and Bob could be on a router close to the edge of the compromised network).

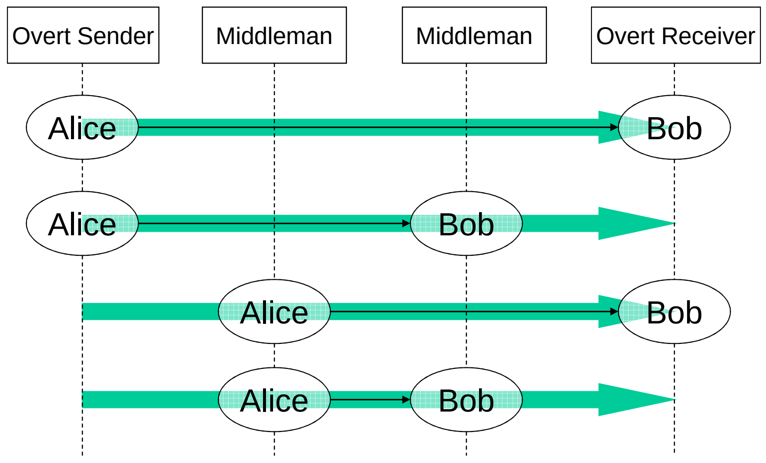


Figure 2.2: Possible combinations of different covert sender/receiver locations

2.1.4 Applications

A diverse range of individuals and groups has found reason to utilise covert channels for communication and coordination. Typically this is motivated by the existence of an adversarial relationship between two parties, such as government agencies versus criminal or terrorist organisations, hackers or corporate spies versus company IT departments or dissenting citizens versus their governments.

Clearly, government agencies, criminals, or terrorist organisations have an interest in keeping their communication secret. However, simply using encryption does not prevent adversaries from detecting communication patterns. Often only the evidence that communication takes place is sufficient to detect the onset of activities, discover organisational structures or justify police warrants.

For example, it has been hypothesised that terrorists use covert channels to coordinate their actions [26]. Recent evidence now confirms that they use information hiding. Published extracts of emails exchanged between terrorists in the UK and Pakistan show that the organisation of planned attacks was disguised as seemingly harmless conversation about ordering goods for a future shop opening [34]. However, this is still simple steganography and not a covert channel in network protocols.

Once spies or hackers have compromised computer systems they usually ex-filtrate data or instrument the systems for malicious purposes, including communication with installed Trojan horses (malicious programs disguised as legitimate software) or tools for launching Denial of Service (DoS) attacks. Such activities generate network traffic that, if not covert, would immediately alert system administrators, who would then also discover the compromised systems. Ex-filtrating sensitive data over covert channels does not even require compromised computers. It is sufficient if the attacker compromises an input device such as a keyboard [35].

Often even ‘ordinary’ users may want to use covert channels to bypass their company or Internet Service Provider (ISP) firewalls in order to access Internet resources. Furthermore, recent attempts by some governments to limit the freedom of speech in the Internet have led to proposals for using covert channels to circumvent these measures [27, 36]. In countries that forbid strong encryption of data, covert channels can be used to ‘secure’ the information transport, although this is not strong security in the cryptographic sense.

Network administrators can use covert channels to ‘secure’ network management related communication by hiding it from hackers [37]. Again this is not strong security in the cryptographic sense. Honeypots, computer systems set up as trap for hackers, can also use covert channels to export logged data in real-time hidden from attackers [38].

Computer viruses or worms can use covert channels to spread themselves undetected or for covertly exchanging information necessary for distributed processing (e.g. execute brute-force attacks on cryptosystems [39]).

Moskowitz *et al.* showed that imperfections in anonymous communications are effectively covert channels usable to thwart anonymisation [40]. Covert channels have been used for breaking anonymisation in multiple scenarios. Xu *et al.* described an attack on traffic trace file anonymisation through covert channels [41] and Bethencourt *et al.* developed a technique to identify the locations of sensors used for detecting malicious network traffic [42]. Murdoch *et al.* developed covert-channel based techniques to reveal servers hidden inside anonymisation networks [43, 44].

Covert channels can also be used for transmitting authentication data. A number of techniques were developed for allowing authorised users to access open firewall ports while presenting these ports as closed to all other users. One particular technique, called “port knocking”, uses covert channels for sending the authentication data [45]. Mazurczyk *et al.* proposed using covert channels and steganography to link control information, including authentication data, to the actual data flows [46, 47].

A number of researchers developed packet traceback techniques using covert channels [48, 49, 50, 51]. Traceback techniques provide downstream nodes with information about the path of incoming packets. This is important in case of DoS attacks, because it allows filtering the attack traffic at upstream nodes and tracing back attackers across intermediate hosts (stepping-stones).

2.1.5 Countermeasures

Before any action can be taken against a covert channel it first needs to be identified. A number of formal methods were developed for identifying covert channels in specifications or implementations of single host systems during the design phase or in an already deployed system (see Section 2.4). Only a few works exist on formal techniques for identifying covert channels in network protocols (see Section 2.4).

Once a covert channel has been identified it can be handled. The generally available countermeasures are:

- *Eliminate* the channel.
- *Limit the bandwidth* of the channel.
- *Detect and audit* the channel.
- *Document* the channel.

If a covert channel was not removed in the design phase the next best option is to eliminate its possible use, because even low-capacity channels could be successfully exploited. However, the removal of all covert channels leads to very inefficient systems, since they can often only be removed completely by replacing automated procedures with manual procedures [52]. Furthermore, covert channels based on the modulation of visible message parameters are inherent in distributed systems, such as computer networks.

Therefore, we and many other researchers believe that covert channels cannot all be completely eliminated [53, 54]. This is also acknowledged by the security standards. For example, the Orange Book treats covert channels with capacities of less than one bit per second as acceptable in many scenarios [19].

If a channel cannot be eliminated its capacity should be reduced. What is an acceptable capacity depends on the amount of information leakage that is critical. For example, if the capacity is so small that classified information cannot be leaked before it is outdated, then the channel is tolerable. Limiting the channel capacity is often problematic, because it means slowing down system mechanisms or introducing noise, which both limit the performance of the system.

Covert channels that are not eliminated or limited should be audited, which requires their reliable detection. Auditing acts as deterrence to possible users and also allows taking actions against actual users. Covert channels with capacities too low to be significant, or which cannot be audited, should at least be documented (e.g. in the protocol specification), so that everybody is aware of their existence and potential threat.

2.1.6 Evaluation criteria

We use three main criteria for evaluating covert channels in network protocols that are similar to those used for evaluating steganographic systems [33]:

- *Capacity* determines the maximum error-free transmission rate of a covert channel [22]. Capacity is typically measured in bits per second, but for network covert channels it can also be expressed in bits per overt packet.

- *Robustness* determines how easily a covert channel is eliminated or its capacity is limited by channel noise, possibly artificially noise introduced by a warden.
- *Stealth* determines how easily a covert channel can be detected by comparing the characteristics of traffic with covert channel and unmodified legitimate traffic.

Our research confirms that capacity, robustness and stealth are conflicting goals. Usually, it is impossible to simultaneously maximise all of them and users have to choose a trade-off that is best for a particular situation. For example, we demonstrate that sending less data improves the stealth and increasing the redundancy of data improves the robustness, but both reduce the capacity. We also show that robustness can be easily improved by increasing the amplitude of the signal, but this reduces the stealth.

2.2 Covert channel techniques

We now give an overview of existing covert channels focusing on channels that are potentially usable for general-purpose communication. We group the existing channels according to a novel taxonomy. We ignore channels in protocols or protocol options that are basically extinct unless their mechanism illustrates a unique or important approach [6].

2.2.1 Taxonomy

We developed a novel taxonomy for classifying covert channels, because existing taxonomies only provided a very coarse classification. Our taxonomy extends previous work, such as the distinction between storage and timing channels [19] or the distinction between noisy and noise-free channels [22]. We classify covert channels based on the following criteria:

- *Storage* vs. *timing* channels: Traditionally covert channels were classified into storage and timing channels [19]. Storage channels involve the writing of object values by the sender and the reading of them by the receiver. Timing channels involve the sender signalling information by modulating the use of resources over time such that the receiver can observe it and decode the information.
- *Predictable* vs. *variable* vs. *random* cover: The cover is the characteristic of the overt traffic into which the covert data is encoded. A predictable cover means there is basically no variation, whereas a variable cover means there is limited variation. A random cover means the cover data is pseudo-random.
- *Noisy* vs. *noise-free* channel: A channel is the communication channel between Alice and Bob. On a noisy channel there are channel errors: *substitutions* (bits

changed with unknown position), *erasures* (bits changed with known position), *deletions* (bits completely lost) and *insertions* (bits inserted) [22]. On a noise-free channel there are no channel errors.

- *Passive* vs. *semi-passive* vs. *active*: In passive channels Alice acts as middleman and uses the existing traffic of unwitting users as cover. In semi-passive channels Alice generates the overt traffic instrumenting real applications, meaning she has only partial control over the overt traffic. In active channels Alice is also the sender of the overt traffic (fake application traffic) and thus has full control of it.
- *Direct* vs. *indirect*: In direct channels the overt traffic that contains the covert data flows directly from Alice to Bob (who both can be middlemen). In indirect channels there are two flows of overt traffic conveying the covert data. The first is between Alice and an unwitting intermediate host and the second is between the intermediate host and Bob.

2.2.2 Direct noise-free storage channels

Most existing covert channels fall into this category, because there are a huge number of possibilities for these channels. Covert data is encoded in underspecified protocol fields and operations or by exploiting semantic ambiguities. While these channel are easy to implement and efficient because there is no channel noise[2], they can be easily detected by their abnormal behaviour and eliminated by protocol normalisation (see Section 2.4).

Unused header fields

Covert channels can be encoded in unused or reserved bits of frame or packet headers. There is great potential for channels if protocol standards do not mandate specific values or receivers do not check for the standard values. Handel *et al.* proposed a covert channel using the unused bits of the IP header's Type of Service (TOS) field (see Figure 2.3) or of the TCP header's flags field (see Figure 2.4) [16].

Kundur *et al.* suggested using the IP header's Don't Fragment (DF) bit as a covert channel [55]. The DF bit can be set to arbitrary values if the sender knows the Maximum Transfer Unit (MTU) size of the path to the receiver and only sends packets of less than MTU size. Hintz proposed transmitting covert data in the TCP Urgent Pointer that is unused if the URG bit is not set [56]. Since the checksum in UDP packets is optional [57], Fisk *et al.* proposed using the presence or absence of it to signal one bit of covert information per packet [58].

[2]Some channels use header fields that are not immutable in the network, but often they are not modified and the channels are effectively noise-free.

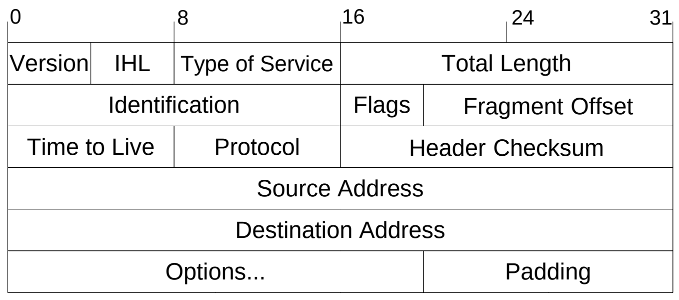


Figure 2.3: IP header structure

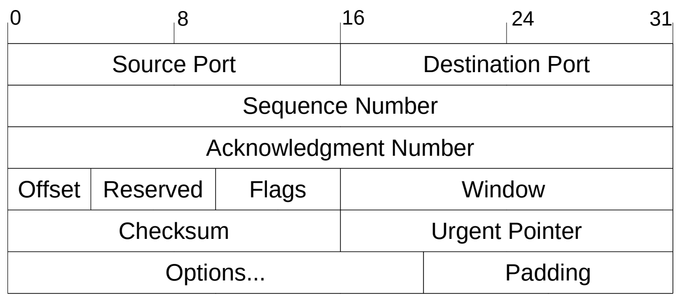


Figure 2.4: TCP header structure

Mazurczyk *et al.* proposed using covert channels to embed control information in Voice over IP (VoIP) flows [46, 47]. VoIP data transmission is usually based on the Real-time Transport Protocol (RTP), and control information is exchanged separately over the Real-time Control Protocol (RTCP) [59]. Instead of using separate RTCP flows Mazurczyk *et al.* proposed embedding the control information into RTP flows. Unused bits in the IP, UDP, and RTP headers signal the type of parameters, and the parameter values are embedded as watermark in the voice data.

Header extensions and padding

Usually there are pre-defined header extensions that allow transporting non-mandatory information on demand, but many protocols also allow header extensions to carry data not foreseen in the original specification, extending the capabilities of the protocol.

Graf proposed transmitting covert data in IPv6 destination options headers that carry optional data for a packet's destination [60]. Covert bits are encoded as option data, and the option type is set so that the overt receiver ignores the option. Lucena *et al.* identified covert channels in the IPv6 hop-by-hop, routing, fragment, authentication and encapsulating security payload extension headers [61]. Trabelsi *et al.* proposed to hide covert data masked as IP addresses in IP route record option headers [62].

Covert information can be encoded in frame or packet padding. For example, Ethernet frames must be padded to a minimum length of 60 bytes. If the protocol standard does not mandate specific values for the padding bytes, any data can be used [63, 16]. Padding

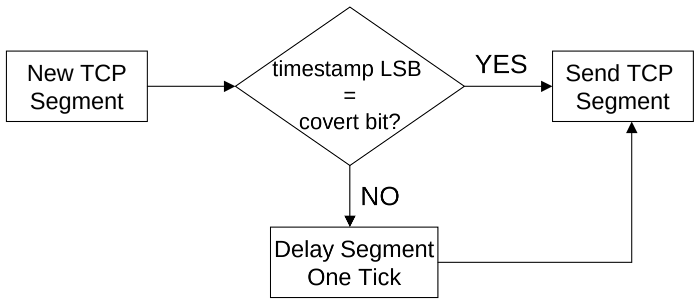


Figure 2.5: Modulating the least significant bit of the TCP timestamp field

of the IP and TCP header to four-byte boundaries (in case header options are present) and padding in IPv6 can also be used to transmit covert data [58, 61].

Modulating timestamp fields

Handel *et al.* noted that covert data can be encoded in IP timestamp header extensions [16]. Giffin *et al.* developed a method for covert messaging through TCP timestamp header options, which are widely used to improve TCP performance [64]. Covert information is inserted in the low order bits of the sender timestamps, because these are assumed to be random for slow TCP connections.

Instead of directly modifying timestamps the algorithm slows the TCP stream so that the timestamps on packets are valid when they are sent. The algorithm compares the least significant bit (LSB) of the timestamp of every TCP segment generated with the current covert bit to be sent. If the LSB matches the covert bit the TCP segment is sent immediately otherwise it is delayed for one timestamp tick (see Figure 2.5).

Modulating address fields and packet lengths

All communication protocols use address fields to identify senders and receivers. The most prominent today are arguably the IP source and destination address fields.

Padlipsky *et al.* and Girling proposed either encoding information in destination address fields directly or by modulating the order of valid destination addresses in subsequent transmissions [65, 15]. The capacity depends on the number of different addresses a covert sender can use. The initial proposals were targeted at link layer frames but the technique could be used to modulate IP addresses or port numbers. Covert information can also be transmitted in source addresses, if they can be modulated [61]. This is the case for IP addresses (if spoofing is possible) or port numbers.

Galatenko *et al.* proposed to send covert information by reordering packets so that destination addresses in a series of subsequent packets are ordered [66]. The covert sender encodes a logical one as a sequence of packets with increasing addresses and a logical zero as a sequence of packets with decreasing addresses. The sequence length used depends on the desired error rate of the channel.

Feamster *et al.* proposed Infranet – a framework to use covert channels in HTTP to circumvent censorship [27]. Web servers participating in Infranet receive covert requests for web pages encoded as a sequence of HTTP requests to harmless web pages and return the content hidden inside harmless images (steganography). Bowyer proposed a very similar mechanism to communicate with Trojans behind firewalls [67]. A Trojan sends HTTP requests to a web server with covert data encoded as URL parameters. The web server returns innocent looking web pages with images containing hidden data (steganography).

Most protocols use length fields to indicate the length of headers, header extensions or messages (frames, packets). Padlipsky *et al.* and Girling proposed to modulate the lengths of link-layer frames to transmit covert information [65, 15]. The same technique can be used to modulate the size of IP packets [61, 68, 69].

Perkins developed a covert channel that encodes the information in the sum of all bits of a message [70]. Covert sender and receiver agree on the maximum possible sum S (all bits set in a message of maximum length) and a division of $[0,S]$ into n intervals. The covert sender encodes covert bits by constructing or re-ordering messages so that the bit sum is in the desired interval. The channel capacity is $\log_2 n$ bits per message.

Various header fields

Marone discovered several covert channels in the Dynamic Source Routing (DSR) protocol used in ad-hoc networks [71]. Covert information can be encoded in header fields present in DSR routing requests, for example the request identification number, hop limit, clock time, or address fields. Covert data can also be piggybacked on regular routing requests in the options header. Another, more sophisticated method presupposes that the covert sender and receiver have a prearranged list of routes, where each route is a symbol. Sending a combination of routes transmits the covert information.

Li *et al.* described a number of covert channels in the Ad-hoc On-Demand Distance Vector (AODV) protocol [72]. Covert information can be transmitted by manipulating the source sequence number field or the destination ID in route requests, or by manipulating the lifetime field in route replies sent by an intermediate node.

Qu *et al.* developed a covert channel based traceback mechanism for IEEE 802.11 Wireless LANs (WLANs) to improve resistance against DoS attacks [49]. In their approach access points encode the path of frames in the More Fragments bit of the Frame Control field, and the Duration/ID field. Krätzer *et al.* proposed to embed covert information in various IEEE 802.11 header fields, such as the Retry bit and More Data bit of the Frame Control field, and the Duration/ID field [73].

Dyatlov *et al.*, Kwecka and Van Horenbeeck proposed various methods for embedding covert channels into HTTP protocol headers [74, 75, 76]. These encompass encoding covert data into header field values, the order of header fields, the use of lower or upper case, the presence or non-presence of optional header fields, the use of multiple white spaces, and new non-standard header fields. Castro *et al.* developed a method for transmitting covert information through HTTP cookies [77].

Bai *et al.* proposed a covert channel in the jitter field of the RTCP protocol [78]. The covert sender replaces the least significant bits of actual jitter values with covert bits.

Corrupted frames/packets

Since IEEE 802.11 wireless networks have variable bit error rates, they provide an opportunity for injecting synthetic ‘corrupt’ frames. Szczypiorski *et al.* proposed a covert channel where all stations that are part of the channel communicate via sending some percentage of their frames with covert data and intentionally created bad checksums [79]. Other stations discard the ‘corrupted’ frames.

Butti *et al.* proposed sending covert information across IEEE 802.11 networks in unsolicited ACK frames or invalid frames with deliberately incorrect checksums [80]. The sender encodes covert data in the payload and a magic number inside the receiver address. The receiver decodes the data from frames containing the magic number.

Krätzer *et al.* proposed another covert channel in IEEE 802.11 using corrupted frames [73]. The covert sender encodes bits by duplicating frames of specific connections (frames going from a particular sender to a particular receiver) and the covert receiver decodes the bits by detecting the duplications.

Payload tunnelling

Payload tunnels are covert channels that tunnel one protocol (usually the IP protocol) in the payload of another protocol to circumvent firewalls. Most of these channels do not aim for stealth but rather for maximising the capacity. A variety of tools exist for tunnelling over protocols that are usually not blocked such as ICMP or HTTP [81].

One of the first approaches for tunnelling over ICMP was Loki, which tunnels protocols in the payload of ICMP echo messages [82]. Today many other IP over ICMP tunnels exist (e.g. [83]). Ray *et al.* proposed a covert channel in ICMP echo messages encoded in the ID field and the payload [84]. Another popular method is to tunnel over HTTP. Padgett developed a tool that tunnels SSH over HTTP proxies [85]. Dyatlov and LeBoutillier implemented tools for tunnelling UDP or TCP over HTTP [86, 87].

Several tools exist for tunnelling IP across the DNS protocol [88]. Communication takes place between a client and a fake DNS server. The client sends covert data encoded in the hostnames in DNS requests (hostname lookups). The server returns data in the DNS responses. However, most of the existing tools use DNS records rarely used for legitimate reasons [88]. Nussbaum *et al.* evaluated the throughput of different DNS-tunnel implementations, but did not analyse how difficult it is to detect the channel [88].

Pseudo-random fields

Some covert channels are embedded in cover data that is pseudo-random (including encrypted data). These channels are similar to Simmons' subliminal channels [31].

The IP Identification (ID) header field is used for reassembling fragmented IP packets. The IP standard requires that each IP ID uniquely identifies an IP packet for a certain time period [89]. The IP ID is set to pseudo-random values on newer operating systems if fragmentation is permitted. The Fragment Offset is used to determine in which order fragments need to be reassembled.

Rowland and Ahsan proposed different ways of encoding covert data in IP ID fields [36, 90]. Cauich *et al.* described how to use this channel between middlemen [91]. If an existing packet is not fragmented Alice inserts covert data into the IP ID and Fragment Offset fields and sets a reserved bit in the flags field. This bit marks packets with covert information so that Bob can distinguish between real final fragments, which have the More Fragments bit set to zero, and the covert channel.

TCP sequence numbers are used to coordinate which data has been transmitted and received. The first sequence number selected by a client is called the Initial Sequence Number (ISN). The ISN must be chosen so that the sequence numbers of new incarnations of TCP connections do not overlap with the sequence numbers of earlier incarnations [92].

Rowland and Rutkowska proposed different techniques to encode covert data into the TCP ISN [36, 93]. However, as Murdoch *et al.* pointed out all the ISN channels proposed previously produce a distribution that differs from those of real operating systems [94]. They developed ISN covert channels tailored to Linux and OpenBSD, where the ISN distribution of the covert channel looks normal [94].

Lucena *et al.* developed two covert channels for the Secure Shell (SSH) protocol [33]. The first active channel hides information in the Message Authentication Code (MAC) header present in each packet. The sender either completely replaces the MAC with encrypted covert data, or uses a short MAC padded with encrypted covert data resembling a long MAC. The second channel is passive. The covert sender intercepts SSH traffic and adds an additional fixed-size encrypted message at the beginning of the already encrypted payload. A magic number marks the presence of the covert data. The covert receiver decodes the covert data and removes it, restoring the original packet.

Szczypiorski *et al.* proposed to embed covert data in the RC4 initialisation vector, which is part of the IEEE 802.11 Wired Equivalent Privacy (WEP) mechanism [79].

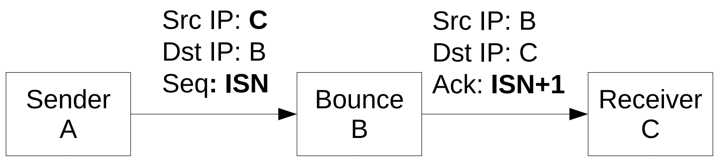


Figure 2.6: The TCP Initial Sequence Number (ISN) bounce channel

2.2.3 Direct noisy storage channels

Noisy storage channels make use of underspecified fields or semantic ambiguities in the same way as noise-free channels, but the data fields used as cover are subject to modifications on the path between Alice and Bob. These modifications can cause errors on the channel, which we also refer to as noise. The noise lowers the capacity, but potentially improves the stealth. Compared to direct noise-free storage channels, there are only very few noisy storage channels and they are much less well understood.

Jones *et al.* proposed using the IP header's TTL field to trace back IP flows without using the source address [48]. Routers change the TTL field of packets so that downstream receivers can unambiguously identify their upstream router. Qu *et al.* and Lucena *et al.* proposed basic schemes for embedding covert information into the TTL [95] and the IPv6 Hop Limit field, which is the IPv6 equivalent of the IP TTL [61].

Since the TTL and Hop Limit fields are modified by network nodes on the path between Alice and Bob and packets can take different paths through the network this channel is noisy. Furthermore, there are other, previously not well documented sources of TTL noise (see Section 3.1).

2.2.4 Indirect storage channels

Indirect storage channels enable Alice and Bob to exchange covert data encoded in protocol fields via an unwitting intermediate node. This increases the stealth because a warden does not see a direct flow of information from Alice to Bob. However, we show that indirect channels are harder to implement and have a smaller capacity than direct channels.

Rowland outlined an indirect channel, called the bounce channel, which is illustrated in Figure 2.6 [36]. Instead of sending a TCP SYN packet with an ISN containing covert data to the receiver directly, the sender sends the TCP SYN packet to a bounce host with a spoofed IP source address set to the intended destination. The bounce host then sends a SYN/ACK or SYN/RST to the receiver with the acknowledged sequence number equal to the ISN+1. The receiver decrements the ACK number to decode the hidden information.

Zelenchuk implemented an indirect IP over ICMP tunnel [96]. The covert sender sends echo requests to a bounce host with spoofed source address set to the address of the covert receiver and the covert data encoded in the payload. The bounce host then sends echo replies to the covert receiver with the same payload.

Danezis proposed an indirect channel using the IP ID field [97]. This channel requires an intermediate host with globally incrementing IP ID counter for outgoing packets, and Alice and Bob must be able to force the intermediary to send packets. In each time interval Alice sends *n* packets to the intermediary, where *n* is the encoded covert data, forcing it to return *n* packets. In each time interval Bob forces the intermediary to send one packet. Bob recovers *n* by computing the IP ID difference of two consecutive packets.

Bounce channels only work if source IP addresses can be spoofed, which more and more networks prevent through ingress filtering. The IP ID channel does not work with recent operating systems, since IP IDs are not sequentially increased anymore.

An anonymous author proposed an indirect covert channel over the DNS protocol that exploits negative caching of domain names [98]. Alice and Bob agree on a series of non-existent domains. Alice recursively queries for all domain names for which she wants to transmit a logical one and does nothing otherwise. Bob non-recursively queries for all domain names interpreting a cached response as one and an uncached response as zero.

Bauer proposed using covert channels in web traffic to enable anonymous communications [99]. The information is hidden in JavaScript/HTML and transported through the use of JavaScript redirects. An observer who cannot look into the content transported by HTTP cannot distinguish between harmless web surfers and covert senders/receivers.

2.2.5 Direct timing channels

Here we discuss channels that encode covert data in the timing of frames, packets or messages exchanged by Alice and Bob directly. Timing channels are always noisy because of timing inaccuracies at the sender/receiver and network jitter mainly caused by varying queuing delays. The capacity of timing channels is often lower than that of noise-free storage channels, but they are potentially harder to detect and eliminate.

Packet rate

Covert information can be encoded by varying packet rates. Alice varies her packet rate between two (binary channel) or multiple packet rates each time interval. Bob measures the rate in each time interval and decodes the covert information. Alice and Bob need a mechanism for synchronisation of the time intervals.

Padlipsky *et al.* outlined a timing channel where the sender either transmits or stays silent in each time interval [65]. This on/off timing channel is a special case of the binary channel where one rate is zero and the other some chosen rate. Girling also identified rate-based timing channels and suggested mitigating the noise problem by adjusting the packet rates [15].

Cabuk *et al.* implemented the on/off timing channel [100]. In their scheme the covert data is divided into small fixed-size frames and synchronisation is achieved through a special start sequence at the beginning of each frame. Cabuk *et al.* noted that their scheme does not entirely solve the synchronisation problem and mentioned better techniques as future work. Recently, Yao *et al.* studied the capacity of such channels based on packet-rate distributions measured in real networks [101].

Luo *et al.* proposed to encode covert data into the length of TCP data bursts, where a data burst is a number of TCP segments sent between two TCP ACK arrivals [102]. The channel is more robust against packet jitter, loss and reordering, but has very low capacity. Furthermore, compared to other timing channels its stealth is low, as covert channels behave very different from normal flows [102].

Inter-packet times

Berk *et al.* introduced a packet-timing channel that does not require synchronisation of time intervals because the covert information is encoded in inter-packet times (inter-packet gaps) of consecutive packets [103]. They compared channels with two gap values (binary channels) and multiple gap vales, and demonstrated a mechanism by which the sender can pick the optimal symbol distribution in multi-symbol channels given the channel characteristics.

Sha *et al.* developed a device that hooks into the connection between keyboard and computer and ex-filtrates all keystrokes by modulating the inter-packet times of network traffic send by the victim [35]. The attack requires physically compromising the victim's keyboard connection.

Gianvecchio *et al.* [104] developed an improved variant of the inter-packet gap timing channel and evaluated its performance. They proposed to fit a model to the inter-packet gap distribution of real traffic and then use the model to generate covert channels with identical distribution. If the inter-packet times of normal traffic are independent and identically-distributed (iid) this channel is very hard to detect. However, as we show in Section 4.1 not all application traffic has iid inter-packet times. In fact a large portion of the traffic we analysed has correlated inter-packet times.

Sellke *et al.* proposed another scheme for encoding covert data in inter-packet times and evaluated the achievable bit rate and error rate based on experiments across the Internet [105]. Similar to [104] they also showed that in theory timing channels can be made indistinguishable from normal traffic by mimicking normal iid inter-packet times.

Liu *et al.* introduced a covert timing channel that encodes the covert data such that the normal distribution of inter-packet times is closely approximated and spreading techniques are used to provide robustness [106]. Their channel is hard to detect with simple shape and regularity tests, however these are known to be insufficient [107].

A number of researchers proposed schemes for embedding watermarks into packet flows by modulating inter-packet times [108, 109, 51]. The main purpose of these techniques is to trace back traffic across proxies, anonymisation networks or stepping stones. We do not discuss the schemes in detail here because they prioritise robustness over stealth and channel capacities are generally very low.

Message sequence timing

Wolf mentioned the possibility of constructing covert channels by modulating the use of protocol operations [63]. For example, a covert receiver can acknowledge each frame separately or wait until two frames have arrived before acknowledging the first. Handel *et al.* proposed a covert channel based on modulating the clear to send/ready to send (CTS/RTS) signals of serial port communication [16]. This technique could be applied to other protocols utilising CTS/RTS, such as WLANs.

Eßer *et al.* implemented a web-based timing channel and analysed its capacity [110]. In their scheme a web server sends covert data to a client by delaying a response (logical one) or responding immediately (logical zero). Li *et al.* described timing channels in the AODV protocol [72]. Alice modulates the times between successive AODV route requests, and Bob decodes the information from the message timing.

Zou *et al.* proposed a technique for embedding covert channels into the File Transfer Protocol (FTP) [111]. Covert data is transmitted through varying the number of FTP NOOP commands [112] send during idle periods; the number of NOOPs sent is equal to the integer value of the covert data.

Packet loss and reordering

Servetto *et al.* demonstrated that channel erasures intentionally introduced at the sender can be used as covert channel [113]. In practice, the technique requires per packet sequence numbers, so the receiver can detect the loss. Erasures are realised by artificially losing packets at the sender.

Mazurczyk *et al.* developed a covert channel utilising packet loss and retransmissions [114]. In their scheme Bob does not acknowledge a successfully received packet. Alice then retransmits the packet, but the payload of the retransmitted data contains a steganogram instead of user data.

Kundur *et al.* described a covert channel implemented through packet reordering [55]. Because a set of n packets can be arranged in any $n!$ ways a maximum of $\log_2 n!$ bits can be transmitted. This approach requires per packet sequence numbers to determine the original packet order.

Chakinala *et al.* proposed a formal model for transmitting information via packet-reordering [115]. They developed several channel and jamming models. Using a game-theoretic approach they modelled the channel as game between covert sender/receiver and jammer, and proved the existence of a Nash equilibrium for the mutual information rate.

Luo *et al.* developed a method that encodes covert information in the order of N packets across X flows [116]. Depending on whether single packets or flows can be distinguished from each other there are various ways of encoding the covert data. If one thinks of flows as urns and packets as balls then these encodings are directly related to the counting problem of drawing N balls from X urns. Luo *et al.* also designed and evaluated a detection algorithm. Recently, Khan *et al.* proposed a similar channel [117].

Atawy *et al.* developed another covert channel based on packet reordering [118]. They used fake IP traffic with sequence numbers embedded in the payload. Since the payload does not look like normal traffic any closer inspection would reveal the covert channel.

Frame collisions

Handel *et al.* proposed exploiting the Ethernet Carrier Sense Multiple Access Collision Detection (CSMA/CD) mechanism [16]. If frames collide in CSMA/CD, a jamming signal is issued and the senders back off a random amount of time. The covert sender jams packets of another user. Then it uses a back-off delay of either zero or the maximum value. Therefore, all frames sent will either lead or lag packets sent by the other user, essentially creating a one bit per frame covert channel. The receiver can recover the information by detecting the collisions and analysing the order of frame arrivals. Bhadra *et al.* proposed a similar jamming channel in the slotted ALOHA protocol [119].

To improve performance of shared medium access, splitting algorithms are used to divide the set of collided senders into smaller subsets and then these subsets retransmit in order. Dogu *et al.* designed a covert channel using the First Come First Serve (FCFS) splitting algorithm [120]. The covert information is conveyed in the number of collisions observed in a collision resolution period. The covert sender controls this number by generating dummy packets and causing additional collisions. The covert receiver passively monitors the channel and keeps track of the collision resolution procedure to extract the covert information. A similar covert channel was studied by Wang *et al.* [50].

Li and Ephremides' transmission scheme uses the covert sender's splitting decisions (which subsets it joins) as carrier of covert data [121]. The covert receiver passively tracks the collision resolution procedure. When it detects a successful transmission from the covert sender it can retrieve past splitting decisions, which is the encoded covert data.

2.2.6 Indirect timing channels

Indirect timing channels also use the timing of packets or messages to transmit covert data. However, unlike direct timing channels there is no direct exchange of timing information between Alice and Bob, which improves the stealth. However, these channels are harder to construct, and only very few proposals exist. Furthermore, we demonstrate that capacity is typically lower than that of direct timing channels.

Hintz described an indirect timing channel using a public server as intermediate host [56]. The covert sender sends a large number of requests to the server or stays silent in each time interval, equivalent to one bit per time interval. The covert receiver periodically probes the server and measures the response time to recover the covert information.

Murdoch developed a channel that is a combination of packet rate and timestamp modulation channels [44]. The channel requires an intermediary that receives and sends packets to both covert sender and receiver. The channel exploits the fact that a host's CPU temperature depends on the number of service requests per time unit it processes and the skew of a host's system clock depends on the temperature. The covert sender either sends requests to the intermediary (logical one) or stays silent (logical zero), thus changing the temperature and indirectly the clock skew. The covert receiver estimates the intermediary's clock skew based on timestamps in packets sent by the intermediary (e.g. TCP timestamps) and decodes the covert bits.

2.3 Covert channels in multiplayer games

Here we discuss covert channels in games with multiple players, related to the work in Chapter 5. Since the existing channels are encoded in moves or strategies played in card or board games or their electronic versions [122, 123, 124, 125], but not embedded inside network protocols, we discuss them in this separate section. We do not cover steganography in single-player games, such as mazes, jigsaw puzzles or Sudoku.

Murdoch *et al.* investigated covert channels for collusion in an online *connect-4* contest where one human contestant could enter multiple programs as players [122]. Murdoch *et al.* won the contest by deploying two types of colluding players: foxes and chickens. Foxes would play their best against other competitors. Chickens would deliberately lose against foxes and play their best against other competitors. Chickens used a covert channel based on redundancy in the moves of the game to detect a fox. Murdoch *et al.* also outlined how the timing of moves could be used to encode covert information [122].

Hernandez-Castro *et al.* proposed a framework for hiding data in games based on game theory [123]. They described how covert information should be hidden in game strategies and explore possible countermeasures. They integrated the proposed covert channel into a Go program and analysed the effectiveness of several detection strategies.

Diehl derived a notion of security for game-based covert channels depending on strategy imitation and performed an information-theoretic analysis to calculate the total amount of information sent during games and the share usable for secure covert channels [124]. They developed and analysed a proof-of-concept channel for a simplified version of poker.

Desoky *et al.* proposed novel methods for concealing messages in chess-related covers, such as training documents, game analysis, and news articles [125]. They developed a proof-of-concept implementation and validated it through steganalysis.

The existing channels have only small capacities. They are useful for collusion, which usually was the main objective, but are not well suited for general-purpose communication. Furthermore, they were all developed for turn-based games and cannot be readily applied to modern real-time network games.

2.4 Countermeasures

Section 2.1.5 summarised the available countermeasures against covert channels. In this section we discuss previous work grouped according to the different countermeasures: identification, elimination, capacity limitation and detection of covert channels.

2.4.1 Identification

Several formal methods were developed for identifying covert channels in specifications or implementations of single host systems. They can identify channels during the design phase, or in an already deployed system. The existing techniques can be grouped into the following categories: information flow analysis [126, 127], non-interference analysis [128], Shared Resource Matrix (SRM) method [129, 130] and Covert Flow Tree (CFT) method [131]. Gligor provides a good introduction to the different methods, except CFT [14]. There are only a few works on formal techniques for identifying covert channels in network protocols.

Donaldson *et al.* discussed how analysis techniques, SRM in particular, could be applied to network protocol covert channels [132]. They proposed analysing network covert channels by separately inspecting host-to-host channels on the lower network layers and intra-host channels between processes on a single host.

Hélouët *et al.* proposed to perform covert channel analysis for distributed systems at the requirement level, when design decisions can still be made to eliminate or limit covert channels [133]. Covert channels detected during the design phase are not implementation-specific, and thus are likely to be present in any implementation. Their approach is based on a representation of requirements by scenarios.

Aldini and Bernardo proposed a method for combining covert channel identification and performance evaluation [134]. The advantage of this integrated approach is that it provides insights into how to trade off quality of service with channel capacity. They applied their methodology to the PUMP model (see Section 2.4.3), obtaining the relation between channel capacity and rate of served connection requests.

2.4.2 Elimination

Host security

Host security cannot remove covert channels, but it can prevent their exploitation in some application scenarios. If hosts were secured from being hacked, the installation of Trojans, and modifications of software or the network stack would be impossible, thus hackers could not exploit covert channels. However, detecting that a host was hacked is difficult if the attacker is skilled. Therefore, relying on host security could be dangerous and it would be better to eliminate covert channels in the first place. Furthermore, this approach does not solve the problem in other application scenarios (e.g. censorship circumvention).

Network security

One approach to counter tunnelling channels is to block protocols or ports that are susceptible to covert channels. For example, ICMP is blocked by many firewalls these days preventing channels such as Loki [82]. Obviously, in the Internet some protocols cannot be blocked because they are vital (e.g. DNS), or because their services are too important (e.g. HTTP). However, in a closed network protocols prone to covert channels could be blocked, or replaced by versions with fewer or limited covert channels.

The leakage of classified information from a high security system to a low security system is prevented by a network design where only hosts on the same security level are allowed to communicate. Such an approach may be practical for highly secure networks, but not for diverse large open networks such as the Internet.

Bouncing covert channels [36] only work if IP address spoofing is possible. Besides solving a number of other security issues, preventing IP spoofing closes such channels (e.g. ingress/egress filtering). Furthermore, securing networks against wiretapping, and securing routers against compromise prevents some covert channels [65].

Traffic normalisation

Many of the channels described in Section 2.2.2 can be eliminated by normalising protocol headers, padding and extensions as described by Malan *et al.* [135], Handley *et al.* [136] and Fisk *et al.* [58] in general, or more specifically for the IPv6 protocol by Lucena *et al.* [61] and ICMP tunnelling by Singh [137]. Traffic normalisation can be performed by end hosts or by network devices, such as firewalls or proxies.

Unused or reserved bits and padding can be dealt with easily by setting them to zero and unknown header extensions can be removed. Some covert channels exploit the fact that certain header fields are not always used and their use is indicated by other header fields. This property can be used for normalisation as well. For example, set the IP ID and Fragment Offset to zero if the DF bit is set, and set the Urgent Pointer to zero if the URG bit is not set. Furthermore, it should be ensured that checksums are always used.

A number of other header fields can be rewritten under certain assumptions. For example, set the DF bit and set IP ID and Fragment Offset to zero if the packet is below the MTU size (assuming the normaliser knows the MTU), rewrite the IP ID (assuming the normaliser can manipulate all fragments), rewrite the TCP ISN, source IP address and source port (assuming the normaliser can keep a mapping between original and new values and modify packets going in the opposite direction accordingly). Some firewalls and Network Address Translators (NATs) already do this. TCP timestamps can also be rewritten (assuming the normaliser is located very close to the source) or the low order bits can be randomised.

The same concepts can be used for eliminating covert channels in application protocols. Schear *et al.* proposed eliminating covert channels in HTTP responses by enforcing protocol-compliant behaviour, restricting usable response headers to a fixed set in a particular order, and verifying response header fields against the corresponding object meta-data and the client's request [138].

2.4.3 Capacity limitation

A prerequisite of determining the efficiency of capacity limitation is that the capacity of the covert channel can be estimated. The capacity depends on the size of the object values (storage channels) or the amount of information encodable in the resources (timing channels) and the speed with which the objects or resources can be modulated.

For noise-free channels it is easy to estimate the capacity. For example, Rowland's channels [36] have a capacity of one byte per overt packet. However, the capacity in bits per second depends on the packet rate of the overt traffic. For noisy covert channels the capacity analysis is more difficult. Usually the capacity is derived based on information-theoretic concepts introduced by Shannon [21].

Millen estimated the capacity of covert timing channels with noise and/or memory [139], while Moskowitz analysed the capacity of discrete, noiseless, and memoryless timing channels [140]. Gray developed an upper bound for the capacity of timing channels when Wei-Mings' fuzzy time [141] is used [142]. Bhadra *et al.* derived the capacity of the frame collision channel for slotted ALOHA [119]. Berk *et al.* studied the capacity of binary and multi-symbol inter-packet gap timing channels [103]. More recently several researchers analysed the capacity of different packet-timing channels [104, 102, 101].

Limit address and length field channels

To limit the capacity of the address field channel described in section 2.2.2 previous research suggested limiting the number of possible addresses [65, 15, 132], which means limiting the allowed host-to-host connections. This may be possible in closed networks, but not in open networks, such as the Internet. For a particular host the sender address should always be fixed (preventing IP spoofing), but the number of destination addresses or source/destination ports can hardly be limited to a small number. Instead of limiting the interactions between hosts, sending dummy packets between random hosts inserts noise into the traffic patterns. Indirect routing achieves the same effect more efficiently [143], but still has significant overhead.

Padding all packets to a common size eliminates the packet length modulation channel discussed in section 2.2.2 [65], but this adds significant overhead, especially for small packets. To increase the efficiency Girling proposed to have a small number of available packet sizes, small enough to limit the capacity appropriately [15]. Anonymisation networks use a fixed packet size to prevent traffic analysis [143], but it is unlikely that the modulation of packet size could be effectively limited in current IP networks.

Limit timing channels

Multiple solutions were proposed to eliminate or at least limit the capacity of the direct timing channels described in Section 2.2.5. Either random noise is introduced to mask the covert channel or the overt channel is forced to use fixed packet or message rates and dummy packets or messages are inserted when useful information is not sent [53]. Wei-Ming's *et al.* fuzzy-time proposal makes all clocks in the system noisy [141]. A sender cannot exactly time outgoing packets and a receiver cannot accurately measure the timing.

Link padding forces a packet flow to adhere to a specific traffic pattern (e.g. packet rate) by delaying packets and injecting dummy packets if necessary and should eliminate packet timing channels [144]. However, Graham *et al.* showed that even if link padding is used information about the source's traffic rate is still leaked because of the inability of the padding gateway to completely isolate the processing of outgoing packets from the interrupt processing necessary to handle incoming packets [145]. These imperfections can still be used as a covert channel.

Because padding links to a single packet rate creates significant overhead, Girling proposed that senders could emit a small number of different packet rates [15]. This increases efficiency and limits covert channels to acceptable capacities.

Message sequence timing channels can be eliminated by buffering and delaying connection attempts or service requests. Spurious data can be inserted into the network against wiretapping receivers, but this does not help against end-host receivers. Schear *et al.* proposed delaying HTTP responses to limit the capacity of HTTP-based timing channels [138].

Giles *et al.* studied the problem of limiting the capacity of timing channels as a game between the covert sender-receiver pair and a jammer [146]. The jammer attempts to retime the packets from the covert sender. The channel capacity is the objective of the game: the jammer wants to reduce it, while covert sender and receiver want it to be high. Giles *et al.* proved the value for certain games and provided corresponding coding schemes.

Liu *et al.* studied how the capacity of their channel can be limited by artificial network jitter [106], but they did not investigate the effects on the application performance. Wang *et al.* proposed to eliminate inter-packet gap timing channels by randomising inter-packet times [147]. The impact of the proposed scheme on UDP or TCP throughput was briefly investigated, but the impact on certain applications remains unclear.

Split connections

One of the simplest common security policies is the Bell-La Padula model [148]. It can be summarised as “no read up and no write down”, which means a low-security entity (low) must not read from a high-security entity (high) and high must not write data to low. A problem arises when low wants to reliably send data to high. Reliable communication requires high to return acknowledgements (ACKs) for the data received and the timing of the ACKs can be manipulated to transmit covert data. A number of methods were proposed for minimising the capacity of this covert timing channel [149].

In the store and forward protocol (SAFP) a gateway sits between low and high (see Figure 2.7). When the gateway receives a packet from low it stores it in a buffer and sends an ACK to low. Then it transmits the packet to high and waits for an ACK. When the ACK is received the gateway removes the packet from the buffer. However, when the buffer is full the gateway must wait for high to acknowledge a received packet until another packet from low can be acknowledged and stored in the buffer; the time it takes the gateway to send an ACK to low is directly related to the time of receiving an ACK from high. High can ensure the buffer is always filled and still exploit the covert channel.

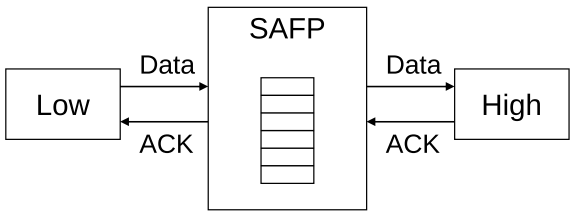


Figure 2.7: The Store and Forward Protocol (SAFP) gateway – a simple approach for limiting the covert timing channel in the flow of ACKs from high to low

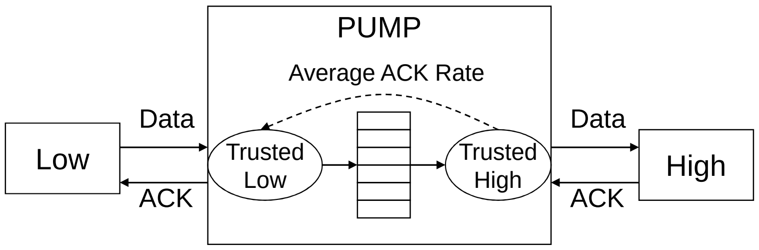


Figure 2.8: The PUMP significantly reduces covert channel capacity of the SAFP because it 'decouples' high's ACKs from ACKs sent to low

The PUMP model substantially reduces the channel capacity of the SAFP [150, 151]. The PUMP uses an historic average of high's ACK-rate as the rate of sending ACKs to low (see Figure 2.8). For every packet from high received by the trusted high process a moving average of high's ACK rate is updated. When the trusted low process receives a message from low it inserts the message into a buffer, and then sends an ACK to low after a delay. The delay is a random variable chosen from an exponential distribution with the mean equal to the current average of high's ACKs rate. Although the PUMP does not completely eliminate the covert channel it significantly decreases its capacity.

2.4.4 Detection and auditing

Auditing requires a reliable detection of covert channels. Many of the covert channels described in Section 2.2 only provide 'security by obscurity' and are easy to detect. All proposed detection methods are based on the detection of non-standard or abnormal behaviour. The assumption is that the warden knows the normal behaviour of protocols and hosts and can detect deviating 'abnormal' behaviour caused by covert channels.

For example, unusually high rates of packet loss or packet reordering (see Section 2.2.5) or frame collisions (see Section 2.2.5) could indicate potential covert channels. Similarly, the SSH middleman covert channel (see Section 2.2.2) could possibly be detected because it changes the packet length distributions of regular SSH connections.

Smith *et al.* derived a general method for detecting storage or timing covert channels based on statistical inference techniques [152]. The probability of traffic being a covert channel is estimated based on deviations between the value distributions of characteristics of the traffic under investigation and regular traffic. More effective techniques for particular channels are described in the following subsections.

However, covert channels that look identical to normal use of protocols are hard to detect. For example, Murdoch's TCP ISN channel has a value distribution matching the distribution of real operating systems [94], and Lucena's SSH MAC channel has statistical characteristics identical to real MACs [33]. The only way a warden could reliably detect these covert channels is to somehow detect the embedding process at the covert sender.

Furthermore, it is difficult to detect covert channels if there is a lot of variation in the usual behaviour. For example, Krätzer's frame duplication channel is potentially hard to detect because usually frame retransmission rates vary significantly [73].

Header field channels

Most protocol standards mandate that unused or reserved bits and padding must be filled with specific values (e.g. zeros). Even if this is not the case the behaviour of actual implementations can be viewed as de-facto standards [94]. All covert channels based on non-standard use of protocols are easy to detect. Furthermore, some proposed covert channels are obsolete because previously unused bits are now used (for example, some bits in the IP header are now widely used for explicit congestion notification), or defined messages or extension headers are de-facto not used anymore and their presence would be suspicious (for example, IP timestamp header extensions).

Other covert channels exploit the fact that some header fields have 'arbitrary' values within the requirements of the standard. However, if the fields are naïvely used and the resulting value distributions are different from the normal distributions generated by operating systems, the covert channels are easy to detect [94]. The usual approach is either to train a classifier on the normal and abnormal behaviour, or to train a classifier on the normal behaviour and detect anomalies. The behaviour is analysed from a set of traffic flows, where each flow is described by a number of characteristics (features).

Sohn *et al.* demonstrated that covert channels with a simple encoding in the IP ID or TCP ISN field [36] are discovered with high accuracy by Support Vector Machines (SVMs) [153]. They evaluated different feature sets and achieved classification accuracies of up to 99%. Tumoian *et al.* analysed the accuracy of a neural network to detect Rutkowska's [93] TCP ISN covert channel [154]. First, the neural network was trained to predict successive ISNs for different operating systems. Then ISNs used by hosts were monitored and compared to the prediction models. An actual ISN sequence not matching any model indicates a covert channel. Tumoian *et al.* found that for more than 100 consecutive ISNs observed the detection accuracy reaches 99%.

Application protocol covert channels can be detected in a similar way as discussed for HTTP in [138, 75].

Timestamp channels

Hintz proposed a detection method for the TCP timestamp channel described in section 2.2.2 [56]. In low-speed networks a randomness test can be applied to the LSB of the timestamps. Too much randomness would reveal the covert channel. In high-speed networks the segment rate is usually larger than the TCP clock's tick rate, which is only between 1 Hz and 1 kHz [155]. A warden can detect the channel by computing the ratio of different timestamps used and the total number of possible timestamps (depending on the duration of the connection). For a normal connection the ratio should be close to one (at least one segment sent every clock tick), but for the covert channel it is close to 0.5 (if a timestamp's LSB is not equal to the covert bit to be sent one clock tick is skipped).

Packet timing channels

Venkatraman *et al.* proposed to audit the change of traffic rates over time to detect packet rate channels [156]. If the traffic rate of one host changes by more than a certain threshold this could indicate a covert channel. They proposed setting the threshold to the standard deviation of the regular rate change observed in the past for a large set of hosts.

Cabuk *et al.* proposed a detection method for on/off packet rate timing channels [100]. They defined a regularity metric that measures whether the cumulative inter-packet time distribution of a traffic flow has only a small number of large jumps (indicating a covert channel) or is more evenly spread (indicating a normal flow). Cabuk *et al.* showed that their technique detects covert channels even if the sender changes the packet rates or there is random noise [100].

Berk *et al.* proposed methods for detecting simple binary or multi-symbol inter-packet gap timing channels [103]. For binary channels the inter-packet times histogram has two distinct spikes, and the mean is between the spikes and has a very low frequency. For normal flows the histogram has a higher frequency at the mean. For multi-symbol channels Berk *et al.* argued that a skilled covert sender would pick a symbol distribution that maximizes the capacity. The warden can also estimate the optimal symbol distribution, compare it to the distribution of the traffic under observation using a similarity test, and detect the presence of the covert channel if both distributions are similar.

A fundamental flaw with this approach is that a covert sender would not choose to maximise the capacity if this compromises the channel. A practical problem is that the warden would have to build the channel matrix for each suspect traffic flow or have a very large number of pre-built channel matrices [103].

Gianvecchio *et al.* proposed new entropy-based metrics to identify different inter-packet gap timing channels [107]. They showed that previous metrics (regularity test and Kolmogorov-Smirnov test) are unable to identify all channels, but a combination of entropy and estimated entropy rate detects all channels.

Stillman proposed to detect timing channels by computing plausible covert bit strings from the inter-packet gaps and scanning for these bit strings in the sender's random access memory [157]. The approach requires that the warden has access to the memory of the covert sender. Also, in practice it is very difficult to compute the bit strings without knowledge of the secret shared by covert sender and receiver and to identify the bit strings in memory if the covert sender uses memory-obfuscation techniques.

Payload tunnelling

Sohn *et al.* used the SVM-based technique described in section 2.4.4 to evaluate the accuracy of detecting covert channels embedded in ICMP echo packets. They achieved classification accuracies of up to 99% when training a classifier on normal packets and abnormal packets generated by Loki [158].

Pack *et al.* proposed detecting HTTP tunnels by using behaviour profiles of traffic flows [159]. The profiles are based on a number of metrics such as the average packet size, number of packets, ratio of small and large packets, change of packet size patterns, total number of packets sent/received, and connection duration. If the behaviour of a flow deviates from normal behaviour it is likely to be an HTTP tunnel.

Borders *et al.* developed a tool for detecting covert channels over outbound HTTP tunnels based on a similar approach [160]. Their tool analyses HTTP traffic over a training period, and is then able to detect abnormal HTTP flows using metrics such as request size, request regularity, time between requests, time of the day, and bandwidth.

2.5 Conclusions

Previous research proposed many different covert channels. But most of the existing channels are simple direct noise-free storage channels that are easy to detect and eliminate in principle, even though in reality appropriate countermeasures may not have been deployed widely yet.

Most covert channels described in Section 2.2.2 use a predictable cover (e.g. unused header fields). While effective against unknowing adversaries they are easy to detect or eliminate. Other channels use pseudo-random data as cover (e.g. the TCP ISN). While such channels are hard to detect, if properly encoded, they are still easy to eliminate. Also, it can be counter-productive to hide covert bits in overt data that potentially looks suspicious to Wendy, such as encrypted data.

More complex channels are harder to detect and eliminate, but usually experience channel noise. For users of communication channels noise is usually 'bad', because it reduces the channel capacity. But for covert channels it also has the benefit of increasing the variability of the cover data. It makes the channel less obvious, if the variability at the source is very small.

The limited work on noisy covert channels has only partially investigated their potential capacity, robustness and stealth. Furthermore, to the best of our knowledge previous work never compared the performance of different types of channels. In this thesis we investigate selected noisy covert channels. We show that often their capacity is high and that they can be made very reliable. Our work also characterises the trade-offs between channel simplicity, capacity and ease of detection and elimination.

We analyse the performance of a noisy covert channel in the TTL field. This channel was identified before [95, 61], but previous research has not analysed its capacity. The encoding schemes proposed in [95, 61] are not all stealthy and cannot be used for passive channels, which is why we develop new improved encoding schemes. Furthermore, previous work has not developed and analysed countermeasures.

Recently several researchers studied timing channels in inter-packet gaps, as they are potentially much harder to detect than simple storage channels and less difficult to implement than other timing channels [103, 104, 102, 105, 106]. The currently stealthiest encoding scheme [104, 105] is only hard to detect if normal inter-packet times are not auto-correlated. However, our analysis shows that in real traffic there often is significant correlation. We propose novel improved encoding schemes that are harder to detect. Furthermore, [104, 105] only investigated active channels, whereas we also develop and analyse passive channels.

Very few works analysed the effectiveness of methods for eliminating covert channels in inter-packet gaps, but they did not study the impact on application performance [147, 106]. Since the warden usually applies the elimination technique to all traffic, most of which does not contain covert channels, it should not have a significant negative impact on the performance of legitimate traffic. We demonstrate that artificial network jitter eliminates the improved channels without reducing application performance significantly.

Indirect covert channels provide additional security since the warden has to track them across intermediate hosts. Unless the warden is located on the intermediate host itself, this is often not trivial. It is hard to find such channels on the network or transport layer. The existing bounce channel and IP ID channel [36, 97] have severe limitations and are unlikely to work in the future. This led to a few proposals of indirect storage or timing channels in the widely used DNS and HTTP protocols [99, 98, 56, 44]. But previous work did not analyse the capacity, robustness or stealth of these channels.

Although a few researchers investigated the use of covert channels in board and card games [122, 123, 124, 125], no previous work exists on covert channels hidden inside network protocols of FPS multiplayer network games. We propose a novel indirect storage channel in multiplayer game traffic that is impractical to eliminate. We develop the basic encoding scheme as well as a tailored mechanism for reliable data transport. We then analyse the throughput of the channel and compare it with an estimate of the channel capacity. We also analyse how difficult it is to detect the channel.

We also investigate Murdoch's indirect temperature-based timing channel [44]. We first develop an improved version of the channel with increased capacity, based on a novel technique that minimises one key source of noise. The channel capacity largely depends on the intermediate host and we estimate the capacity for two example intermediate hosts. We also discuss techniques to eliminate or detect the channel.

Some work exists on estimating the capacity of noisy network covert channels, but it is almost entirely focused on direct packet timing channels [140, 156, 103, 104, 102]. We analyse the capacity of all selected channels. We also consider the effects of packet loss and reordering of the overt traffic on the channel capacity, which were largely ignored in previous research, with the exception of recent work in [102].

Until now only a few researchers have studied the use of ML techniques to detect covert channels [153, 158, 154], and their studies do not cover our selected channels. Furthermore, the ML algorithm we use has not been used before to detect covert channels. A key for the successful use of ML techniques are the features that describe the characteristics of normal traffic and covert channel. While some of the features we use were used previously, others are novel to the best of our knowledge.

Chapter 3
Time-to-live Covert Channels

In this chapter we analyse direct noisy storage channels in the IP Time-to-live (TTL) header field. The TTL field limits the lifetime of IP packets, preventing packets from living forever during routing loops [89]. A packet's TTL is set by the sender and decremented by each network node along the path processing the packet's IP header (e.g. routers). Packets are discarded if their TTL reaches zero while still in transit.

Covert bits are encoded into a TTL value, or a succession of TTL values. Although assumed to be reasonably stable between two end-points, the TTL value is nonetheless subject so some variation, as routers and middleboxes modify the TTL of packets and packets can take different paths through the network. This 'normal' variation causes bit substitutions on the channel; we also refer to this as TTL noise. Furthermore, reordered and lost packets may cause further bit substitutions and bit deletions.

The idea of using the TTL field as covert channel is not new. However, the previously proposed modulation schemes [95, 61], ways of encoding data into the TTL field, are not all stealthy and cannot be used for passive channels. Furthermore, previous work has not analysed the capacity of the TTL channel and has not addressed the problem of transmitting information across the channel reliably. Unlike previous work we also consider the use of the TTL channel as passive channel.

First, we analyse the normal TTL variation based on several traffic traces. The results provide insights into how much the covert channel can modify TTL values without revealing itself and give an indication how large the error rate is expected to be. Then we describe the previous modulation schemes and propose new improved modulation schemes. Next, we propose an information-theoretic model for the channel and derive the channel capacity. Then we develop a protocol for reliable data transport. Finally, we evaluate the performance of the channel.

We emulate the use of different modulation schemes using overt traffic from traffic traces and measure the resulting error rates. Based on the channel model and the measured error rates we compute the capacity of the channel depending on the modulation schemes and network conditions. Depending on the available overt traffic the capacity is at least several tens, but up to over one kilobit per second even with modest packet loss and reordering. Despite the noise the capacity is substantial. Furthermore, the noise masks the covert channel, which would otherwise be trivial to detect.

Table 3.1: Packet traces used in the analysis

Trace	Capture location	Date	Public
CAIA	Public game server at Swinburne University, Melbourne, Australia	05/2005 – 06/2006	no
Grangenet	Public game server connected to Grangenet, Canberra, Australia	05/2005 – 06/2006	no
Twente	Uplink of ADSL access network [165]	07/02/2004 – 12/02/2004	yes
Leipzig	Access router at Leipzig University [166]	21/02/2003	yes
Bell	Firewall at Bell Labs [166]	19/05/2002 – 20/05/2002	yes
Waikato	Access router at University of Waikato, New Zealand	04/05/2005	no

We evaluate the throughput of the developed reliable transport technique for aggregate overt traffic taken from traces under different simulated network conditions. We also analyse the throughput across a real network with different rates of TTL errors, overt packet loss and reordering, using single overt traffic flows as carrier. Comparison of the results with the channel capacity shows that our technique is not optimal, but it provides throughputs of at least 50% of the capacity, except in the case of high reordering. The throughput is up to over hundred bits per second.

3.1 Normal TTL variation

First we analyse ‘normal’ TTL variation (TTL noise). We analyse how TTL varies at small time scales of subsequent packets of traffic flows (series of packets with the same IP addresses, ports, and protocol) and not only focus on variation caused by path changes as previous work [161, 162, 163, 164]. An extended version of this study is in [9].

3.1.1 Datasets and methodology

We use packet traces of different size, origin and date for our analysis (see Table 3.1). The CAIA trace contains only game traffic arriving at a public game server, the Grangenet trace contains game and web traffic arriving at a specific server, and the other traces contain a mix of bidirectional traffic.

Usually analysis of network traffic is based on packet flows. Because we found the number of TTL changes is correlated with the number of packets and the duration of flows, we analyse many characteristics of TTL changes based on packet pairs, defined as two subsequent packets of a flow. This isolates the characteristic under study (e.g. estimated hop count) from correlated flow properties (e.g. size and duration). However, for analysis requiring a sequence of packets we still use flows (e.g. in Section 3.1.4).

Table 3.2: Flows and packet pairs with/without TTL changes and percentage of changes

	Flows			**Packet pairs**		
	No change (kilo)	**Change (kilo)**	**Change (%)**	**No change (Mega)**	**Change (kilo)**	**Change (%)**
CAIA	128.6	2.8	2.1	1 456.3	340.0	0.02
Grangenet	283.0	8.6	2.9	215.7	62.1	0.03
Twente	1 354.6	24.7	1.8	95.5	74.8	0.08
Waikato	1 255.9	57.8	4.4	21.2	86.4	0.4
Bell	899.8	52.9	5.6	36.4	87.1	0.2
Leipzig	7 155.1	429.1	5.7	365.5	1 822.7	0.5

First, packets were grouped into unidirectional flows according to IP addresses, port numbers and protocol (each being a series of packet pairs). We only considered flows with at least four packets (‘minimum’ TCP flow) and an average packet rate of at least one packet per second. We extracted the series of TTL values from the IP headers. If different TTL values occur in a flow or between packet pairs this is referred to as TTL change. Otherwise we refer to the TTL being constant.

3.1.2 Amount of TTL variation

Table 3.2 shows the number and the percentage of flows and packet pairs with and without TTL changes (numbers are rounded). Overall, the TTL is constant for the majority of flows, but 2–6% of flows do experience TTL changes. The percentage of packet pairs with TTL changes is between 0.02–0.5%. This shows that if there is TTL variation in flows, changes occur only for a very small number of the flows’ packet pairs.

3.1.3 Distinct values and change amplitudes

Figure 3.1 shows the cumulative density functions (CDFs) of the number of distinct TTL values per flow. Most flows with TTL variation have only two distinct TTL values. Only less than 10% of the flows have more than two values and flows with more than five different TTLs are very rare, except in the CAIA trace.

We examined CDFs of the amplitude of TTL changes of packet pairs, where the amplitude is defined as the difference between the maximum and the minimum TTL value. The amplitude is one for many packet pairs, but in some traces large numbers of packet pairs have amplitudes around 64, 127, or 191 [9]. The reason for these high amplitudes is middleboxes, such as firewalls or proxies, (re)sending packets part of the TCP handshake or teardown on behalf of end hosts (e.g. for SYN flood attack protection).

The TTLs in these packets are set to the initial TTL of the middlebox, which can differ from the initial TTL used by the host. Different operating systems use different

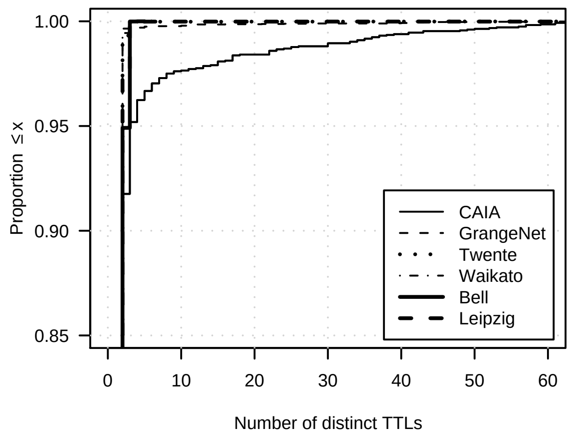


Figure 3.1: Distribution of the number of distinct TTL values per flow

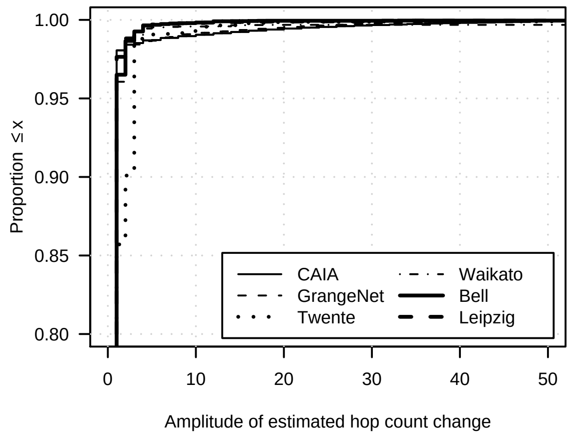


Figure 3.2: Distribution of the estimated hop count amplitude of packet pairs

initial TTLs, the most common being: 64 (Linux, FreeBSD), 128 (Windows), 255 (Cisco) [167, 168]. Therefore, the difference of two TTLs is 64, 127, or 191 plus/minus the number of hops between middlebox and host. Because of this effect TTL changes are more likely at the start or end of TCP flows.

Figure 3.2 shows the CDFs of the amplitude of the estimated hop counts of packet pairs, defined as the difference between estimated maximum and minimum hop counts. The hop count is estimated by subtracting a packet's TTL value from the closest initial TTL. For most packet pairs with TTL changes the hop count changes only by one.

3.1.4 Frequency of changes

Figure 3.3 shows CDFs of the number of TTL changes per flow. For most datasets the majority of flows only have very few TTL changes. But a large percentage of flows in

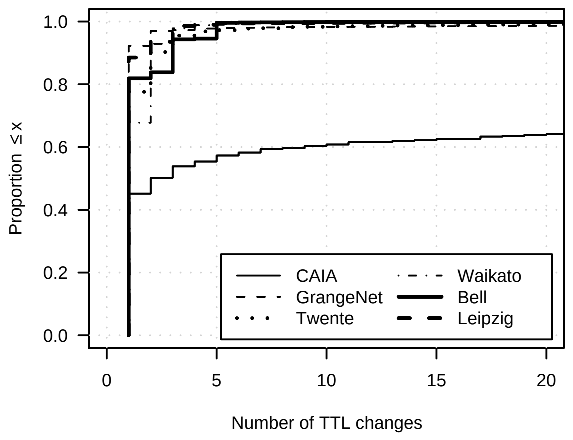


Figure 3.3: Distribution of number of TTL changes per flow (x-axis limited to 20 changes)

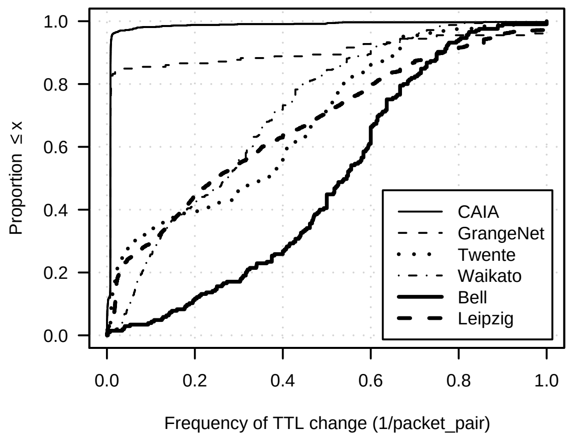


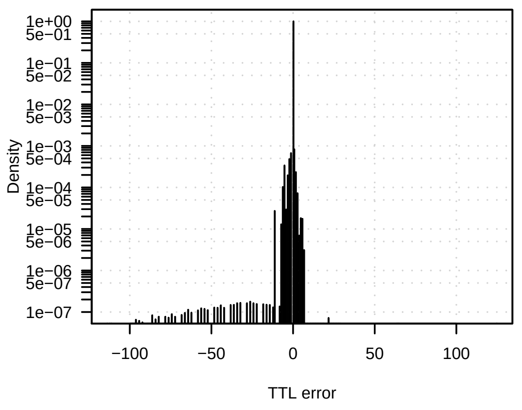
Figure 3.4: Distribution of frequency of TTL changes for flows with at least six TTL changes

the CAIA trace have a large number of changes, because the trace contains many long flows that have roughly periodic TTL changes of unknown origin. Shorter flows are predominant in all other traces.

Figure 3.4 depicts the CDFs of the change frequency for flows with at least six TTL changes. The TTL change frequency of a flow is defined as the number of TTL changes divided by the number of packet pairs. CAIA and Grangenet traces have very low frequencies. Twente, Leipzig, Waikato and Bell have higher frequencies, with roughly half of the flows changing TTL on average every third to second packet pair.

3.1.5 Error probability distribution

We define a *TTL error* as deviation of the TTL value of a packet from the most common value of the TTL during the life of a flow. Let the most common TTL value be $\mathrm{TTL}_{\mathrm{norm}}$.



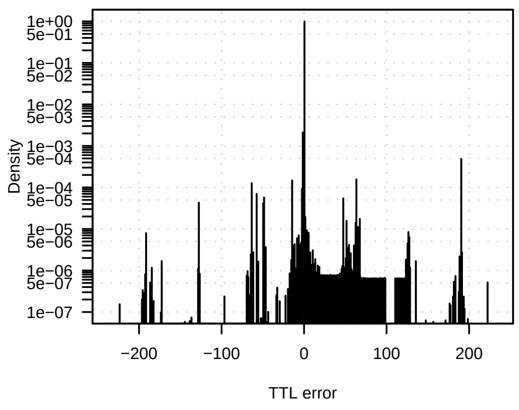


Figure 3.5: TTL error distribution for the CAIA trace (left) and Leipzig trace (right)

Then for a packet i of a flow a TTL error occurs if $\mathrm{TTL}_i \neq \mathrm{TTL}_{\mathrm{norm}}$. We computed the error probability distribution based on the trace files.

Figure 3.5 shows the TTL error distributions for the CAIA and Leipzig traces (other graphs are in Appendix B.1). The average error probability for CAIA is only 0.02% compared to 0.5% for Leipzig (see Table 3.2). Note that the y-axis is logarithmic and we only show error rates above 1^{-7}.

Error values are largely confined between −200 and 200, and the error probability does not monotonically decrease with increasing TTL error. For datasets containing TCP traffic there are the characteristic peaks around ±64, ±128 and ±191 described earlier. The error probability distributions vary significantly between traces and the empirical distributions cannot be easily modelled with standard statistical distributions.

3.1.6 Conclusions

Overall the amount of TTL variation is relatively small. Less than 1% of the packet pairs and less than 6% of the flows experience TTL changes. This provides a good opportunity for TTL covert channels. Normal TTL variation is common enough to not raise suspicion, but not frequent enough to cause high error rates on the channel.

Most normal flows with TTL changes have only two distinct TTL values with a hop count difference of one and there are only a few transitions between different TTLs. This means to avoid detection the covert channel should generally only use two different TTL values that differ by one and avoid very frequent changes.

Most TTL changes are of deterministic nature, meaning the changes occur in specific packet pairs of a flow. For example, in many TCP flows packets part of the TCP handshake or teardown have TTL values that differ from the other TTLs in the flow (see Section 3.1.3). However, there are also flows with approximately periodic changes, flows with infrequent random changes (possibly route changes or anomalies) and flows with frequent random TTL changes (possibly load balancing or route flaps). A more detailed discussion is in [9].

This variety makes it more difficult for covert channels to mimic normal change patterns well. On the other hand it also makes it potentially more difficult for the warden to detect abnormal flows caused by the covert channel.

3.2 Modulation schemes

At the heart of the covert channel is the modulation scheme that defines how covert bits are encoded in TTL values. We first present previously proposed modulation schemes and discuss their shortfalls. Then we present our novel improved modulation schemes. Finally, we discuss implementation issues.

3.2.1 Existing techniques

We group the existing modulation techniques into three classes:

- *Direct encoding* encodes bits directly into bits of the TTL field.
- *Mapped encoding* encodes bits by mapping bit values to TTL values.
- *Differential encoding* encodes bits as changes between subsequent TTL values.

Qu *et al.* described two techniques [95]. The first technique encodes one covert bit directly into the least significant bit of TTL values. Because this potentially increases the original TTL values we refer to the scheme as Direct Encoding Increasing (DEI). The second method encodes bits into TTLs using mapped encoding. The original TTL value represents a logical zero and a TTL value increased by an integer Δ represents a logical one (see Figure 3.6). We refer to this technique as Mapped Encoding Increasing (MEI).

Lucena *et al.* proposed modulating the IPv6 Hop Limit field (TTL equivalent in IPv6) using differential encoding [61]. A logical one is encoded as TTL increase by Δ and a logical zero is encoded as TTL decrease by Δ (see Figure 3.7). Because there is no limit on how much the original TTL value can change we refer to this scheme as Differential Unbounded (DUB).

Qu and Lucena both proposed encoding information by increasing the original TTL value. This is problematic for passive senders because it violates the IP standard [89] and would cause problems if routing loops occur. It also means these techniques cannot be used if the original TTL value already is the maximum value (some operating systems use an initial TTL of 255 [167, 168]). Increasing an already high TTL value would cause the

TTL to ‘wrap-around’ to a very low value in the 8-bit number space. It is then very likely that packets will be discarded before they reach their intended destination.

DUB is problematic because there is no limit on how much TTL values are changed. Long series of zeros or ones lead to large decreases or increases including wrap-arounds. Regardless of the initial TTL value it is likely that some packets are discarded or packets could stay forever in the network during routing loops. The problem can be prevented by limiting the series length using run length encoding or scrambling. Still a warden could easily detect the channel because the modified flows likely have more than two distinct TTL values, which is uncommon for normal flows as discussed in Section 3.1.

3.2.2 New techniques

We propose several new improved modulation schemes. Direct Encoding Decreasing (DED) directly encodes covert bits into TTLs, but the TTL values are always decreased. More than one bit can be encoded per packet, making the scheme tuneable towards capacity or stealth. The maximum number of bits that can be encoded per packet is:

$$n_{\max} = \lfloor \log_2 (I - h_{\max}) \rfloor \;, \tag{3.1}$$

where I is the original TTL at the covert sender, $h_{\max}$ is the upper bound on the number of hops between covert sender and overt receiver and $\lfloor . \rfloor$ denotes the floor operation. The sender encodes covert information by setting the TTL to:

$$\mathrm{TTL}_{\mathrm{S}} = \mathrm{TTL} - ((\mathrm{LSB}(\mathrm{TTL}, n_{\max}) - b) \bmod 2^{n_{\max}}) \;, \tag{3.2}$$

where $\mathrm{LSB}(\mathrm{TTL}, n_{\max})$ is the integer value of the least significant $n_{\max}$ bits of the original TTL and b is the integer value of $n_{\max}$ bits of covert data. Assuming a packet traverses h hops the TTL at the receiver is $\mathrm{TTL}_{\mathrm{R}} = \mathrm{TTL}_{\mathrm{S}} - h$. The covert data is decoded as:

$$b = \mathrm{LSB}(\mathrm{TTL}_{\mathrm{R}} + h_{\mathrm{R}}, n_{\max}) \;, \tag{3.3}$$

where h_{R} is the hop count known by the receiver and without channel errors $h_{\mathrm{R}} = h$.

Mapped Encoding Decreasing (MED) encodes the covert information using two symbols: *low-TTL* signals a logical zero whereas *high-TTL* signals a logical one. Low- and high-TTL are two particular TTL values. The covert sender uses either the default initial TTL (if also the overt sender) or the lowest TTL of the intercepted packets (if a middleman) as high-TTL, and high-TTL minus one as low-TTL. The receiver decodes packets with the higher TTL as logical one and packets with the lower TTL as logical zero. Figure 3.6 compares the MED and MEI schemes.

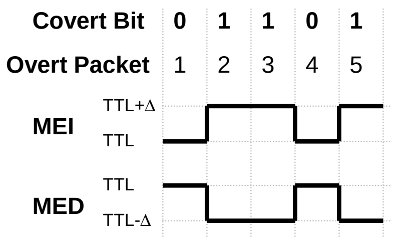


Figure 3.6: Comparison of mapped encoding schemes for an example series of covert bits

Table 3.3: AMI encoding: current TTL based on covert bit and previous TTL

Covert bit	Previous TTL	Current TTL
0	TTL	TTL
0	TTL – Δ	TTL – Δ
1	TTL	TTL – Δ
1	TTL – Δ	TTL

Our last scheme is a differential encoding scheme similar to Alternate Mark Inversion (AMI) coding. Hence we refer to it as AMI scheme. It can be tuned towards stealth or capacity by increasing/decreasing the amplitude of the signal, but it can encode only one bit per overt packet pair. It has the following advantages over DUB: TTL values are always decreased and the TTL values never change by more than one if $\Delta = 1$.

The covert sender encodes a logical zero by repeating the last TTL value. A logical one is encoded by a TTL change, alternating between the two possible values (see Table 3.3). The receiver decodes a constant TTL as logical zero and a TTL change as logical one. Figure 3.7 compares the DUB and AMI schemes.

Decrementing the original TTL eliminates the wrap-arounds and the risk of packets stuck in routing loops. It is still very likely that packets reach their final destination since modern operating systems use initial TTL values of at least 64 [167, 168] and the maximum number of hops between two hosts in the Internet is typically less than 32 [167]. Even with the maximum number of hops increasing in the future there is clearly enough headroom.

Bit error probabilities can be computed for all schemes based on the distribution of TTL changes (see Appendix B.2). However, as we showed in Section 3.1 the TTL error distribution varies significantly between traces and cannot be easily modelled. Therefore, we use emulation to compute the actual error probabilities for each trace (see Section 3.5).

3.2.3 Implementation considerations

If the covert sender is a middleman and encodes covert data into multiple overt traffic flows, for mapped and differential encoding it must encode into each flow separately considering the original TTL values of each flow. Otherwise drastic changes of TTL

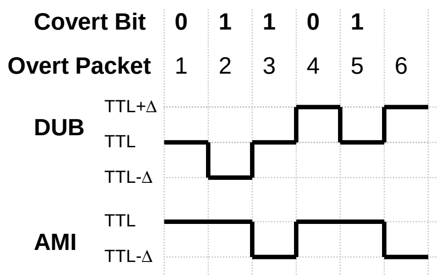


Figure 3.7: Comparison of differential encoding schemes for an example series of covert bits

values would reveal the covert channel. This also means covert sender and receiver need to maintain per-flow state.

Mapped encoding schemes are problematic in this scenario, because usually the receiver does not know the per-flow mapping of bits to TTL values a-priori. However, the receiver must know the mapping prior to decoding. For short flows the receiver may not be able to learn the mapping before a flow ends. If the overt traffic consists of many short flows, which is typical for aggregate Internet traffic, a high error rate is likely.

For binary channels it is sufficient to learn the TTL value for either bit. Therefore the problem is mitigated by sending one special logical zero at the start of each flow. The receiver uses this bit to learn the mapping and then silently drops it. We refer to the modified mapped encoding schemes as MEI0 and MED0.

No flow state is necessary for direct encoding schemes. However, they require that the receiver knows or periodically measures the number of hops between covert sender and receiver. Alternatively, the receiver could guess the hop count assuming it can determine whether the decoded information is valid (e.g. using checksums).

3.3 Channel capacity

We now propose an information-theoretic model for the TTL channel and derive the channel capacity. The capacity is the maximum rate at which communication with arbitrary small error is possible [22]. Our model is not limited to the TTL channel, but could be used for other storage channels in the IP protocol.

We model the TTL covert channel as discrete memoryless channel assuming the current output of the channel only depends on the current input and the error, but not on previous input. We assume a binary channel with one bit of covert data encoded per packet (direct, mapped) or packet pair (differential). We assume the covert data is uniform random (same probability for zeros and ones), which is the case if Alice uses encryption.

The capacity of a channel is affected by channel errors. There are three possible sources of errors for the TTL covert channel:

- bit substitution errors caused by normal TTL variation,

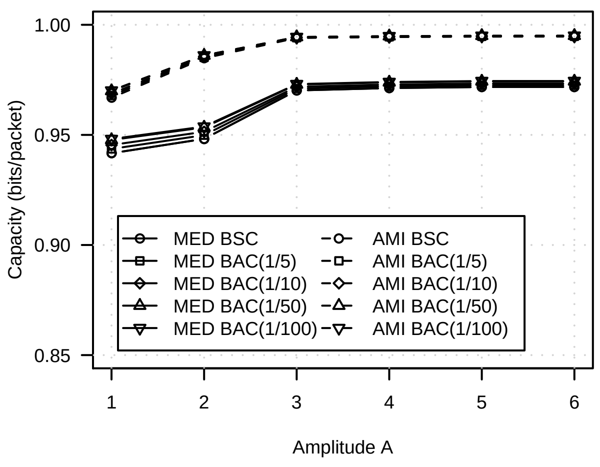


Figure 3.8: Capacity of BSC and BAC with varying degree of asymmetry for the MED and AMI modulation schemes (average error rate across all traces)

- deletions of bits caused by loss of overt packets and
- bit substitution errors caused by reordering of overt packets.

The noise caused by modification of the TTL field on the path between covert sender and receiver and path changes (see Section 3.1) causes bit substitutions on the channel. Whether the TTL noise is symmetric or asymmetric depends on the modulation technique and the error probability distribution.

We model the channel with only TTL noise either as binary symmetric channel (BSC) [22] or binary asymmetric channel (BAC) [169]. The BSC is a channel with two input/output symbols where each input symbol is changed to the other with error probability p. The BAC has two input/output symbols where the first symbol is changed to the second with probability a and the second symbol is changed to the first with probability b. However, the capacity difference of BSC and BAC is small even for larger asymmetries given the typically relatively small TTL error rates.

The overall error rate of BAC and BSC is identical when $p = \frac{a+b}{2}$. If x defines the degree of asymmetry then $a = 2p \cdot x$ and $b = 2p(1-x)$. Figure 3.8 shows an example of the capacity of BSC and BAC with varying x for the MED and AMI modulation schemes averaged across all traces (see Section 3.5.2). The capacity difference between BSC and BAC is less than 0.03 bits per overt packet or packet pair, even for higher asymmetries than observed across all experiments. Also, the capacity of the BSC is always a lower bound for the capacity of the BAC. Therefore, we use the simpler BSC.

How to model the impact of packet loss and reordering on the channel depends on whether the overt traffic supports the detection and/or correction of packet loss and reordering (e.g. retransmissions), assuming the related protocol information (e.g. sequence

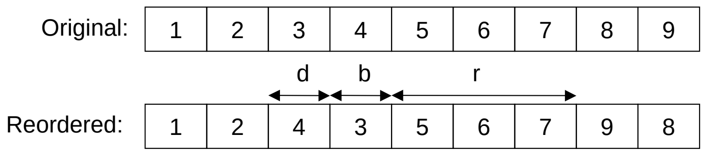


Figure 3.9: Packet reordering model parameters for an example series of packets

numbers) is accessible for covert sender and receiver. It also depends on whether the channel is encoded into one or multiple overt flows.

If the protocol has no sequence numbers (e.g. UDP), the covert receiver does not know which bits were lost on the channel. Hence we model packet loss as a binary deletion channel [22]. If the protocol has sequence numbers (e.g. RTP or TCP) the covert receiver knows which packets were lost, and we model this case as binary erasure channel [22]. If the protocol uses retransmissions, such as TCP, for a single overt flow all packets are eventually (re)transmitted and consequently there are no erasures. However, if covert data is encoded into multiple flows deletions may still occur. There is no guarantee that a TCP flow ends with a proper teardown sequence and there may be packets at the end of flows seen only by the covert sender but not the receiver.

Packet reordering results in substitution errors if the overt traffic has no sequence numbers allowing the covert receiver to process the received packets in the correct order or if the covert channel is encoded in multiple simultaneous flows. We model packet reordering as BSC. For the BSC we need to express packet reordering as error probability. However, this does not preclude the use of a more elaborate model for packet reordering. We use the model proposed by Feng *et al.* that characterises reordering based on [170]:

- average number of packets between reordering events r,
- average reordering delay in packets d and
- average reordering block size in packets b.

Figure 3.9 shows an example sequence of reordered packets and the corresponding model parameters. For mapped and direct encoding schemes only bits in packets that are reordered are affected. However, for differential schemes any packet pair is affected where at least one packet is reordered, meaning there are two more potential bit errors per reordering event. Assuming uniform covert data on average every second reordered bit is wrong. Then the average error probability is:

$$p_{\mathrm{R}} = \begin{cases} \frac{1}{2}\frac{d+b}{d+b+r}, & \text{mapped, direct} \\ \frac{1}{2}\frac{d+b+2}{d+b+r}, & \text{differential} \end{cases} . \tag{3.4}$$

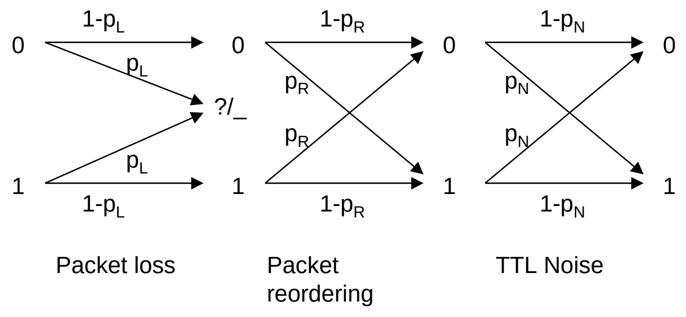


Figure 3.10: TTL channel model

We model the overall channel as a cascade of the three separate channels where the leftmost channel is either a deletion channel with a symbol lost indicated by a "_" or an erasure channel with an unknown symbol value indicated by a "?" (see Figure 3.10).

In the remainder of the section we derive the capacity of the overall channel. The capacity of the BSC is [22]:

$$C = 1 - H(p) = 1 + p \cdot \log_2(p) + (1-p) \cdot \log_2(1-p) \ , \tag{3.5}$$

where $H(.)$ is the binary entropy. The two cascaded BSCs with error probabilities p_R and p_N can be replaced by a single BSC with error probability:

$$p_\mathrm{RN} = p_\mathrm{R}(1-p_\mathrm{N}) + p_\mathrm{N}(1-p_\mathrm{R}) \ . \tag{3.6}$$

The exact capacity of (cascaded) deletion channels is not known, but various lower and upper bounds exist [171, 172, 173, 20, 174]. Diggavi and Grossglauser proved a lower bound of the capacity of a combined deletion/substitution channel depending on the probabilities for deletions p_d and substitutions p_e [175]:

$$C \geq \max\{0, 1 - [H(p_\mathrm{d}) + (1-p_\mathrm{d})H(p_\mathrm{e})]\} \ . \tag{3.7}$$

This bound is tighter than the more general lower bounds for the capacity of deletion/insertion/substitution channels given by Gallager and Zigangirov [171, 172, 173]. This means in any case the lower bound of the capacity of the TTL covert channel is:

$$C \geq \max\{0, 1 - [H(p_\mathrm{L}) + (1-p_\mathrm{L})H(p_\mathrm{R}(1-p_\mathrm{N}) + p_\mathrm{N}(1-p_\mathrm{R}))]\} \ . \tag{3.8}$$

If the overt traffic has sequence numbers we model packet loss as erasures. Depending on the probability of erasures ε and substitutions p_e the cascade of erasure and BSC channel has a channel capacity of [176]:

$$C = (1-\varepsilon)(1 - H(p_\mathrm{e})) \ . \tag{3.9}$$

Table 3.4: Channel capacity based on overt traffic

	UDP w/o seq numbers	**UDP with seq numbers**	**TCP**
Single flow	Equation 3.8	Equation 3.10	Equation 3.12
Multiple flow	Equation 3.8	Equation 3.11	Equation 3.13

When encoding into single flows with available sequence numbers packet reordering does not cause errors ($p_\mathrm{R} = 0$) and the capacity of the TTL covert channel is:

$$C = (1 - p_\mathrm{L})(1 - H(p_\mathrm{N})) \ . \tag{3.10}$$

However, if covert data is encoded in multiple simultaneous flows there may be reordering of packets between flows that cannot be detected, since sequence numbers work only on a per-flow basis. Assuming no abrupt flow ends the capacity is:

$$C = (1 - p_\mathrm{L})(1 - H(p_\mathrm{R}(1 - p_\mathrm{N}) + p_\mathrm{N}(1 - p_\mathrm{R}))) \ . \tag{3.11}$$

If the overt traffic has sequence numbers and is based on a reliable transport protocol ($p_\mathrm{R} = 0$, $p_\mathrm{L} = 0$) we model the channel as BSC and the capacity is:

$$C = 1 - H(p_\mathrm{N}) \ . \tag{3.12}$$

Again, when encoding in multiple flows and assuming no deletions caused by abrupt ends of TCP flows, the capacity reduces to:

$$C = 1 - H(p_\mathrm{R}(1 - p_\mathrm{N}) + p_\mathrm{N}(1 - p_\mathrm{R})) \ . \tag{3.13}$$

Table 3.4 summarises the channel capacity based on single vs. multiple flow encoding for the unreliable UDP and reliable TCP transport protocols.

The capacity C is always in bits per overt packet or packet pair (bits per symbol). If f_S is the average frequency of packets or pairs per second the maximum transmission rate R in bits per second is:

$$R = C \cdot \frac{1}{f_\mathrm{S}} \ . \tag{3.14}$$

For differential schemes packet loss does not only cause deletions or erasures but also substitution errors in each bit following a deletion/erasure. For example, if the AMI scheme is used to send the bit sequence “1111” in five overt packets and the third packet is lost the received bit sequence is “101”. The probability of the additional substitution errors depends on the modulation scheme. Considering all sequences of three bits it is easy to show that for AMI the probability is $\frac{1}{2}p_\mathrm{L}$ for uniform covert data. We model this error as another BSC and replace p_R above with the error rate $\tilde{p}_\mathrm{R}$ of the combined BSC:

$$\tilde{p}_{\mathrm{R}} = p_{\mathrm{R}}\left(1 - \frac{p_{\mathrm{L}}}{2}\right) + \frac{p_{\mathrm{L}}}{2}\left(1 - p_{\mathrm{R}}\right) . \tag{3.15}$$

3.4 Reliable data transport

The maximum transmission rates based on the channel capacity can only be achieved with optimal encoding schemes. Here we first discuss encoding strategies and then present two schemes for reliable data transmission. Our motivation is to demonstrate and evaluate a reliable data transport over the TTL channel, which has a complex non-stationary bit error distribution (see Section 3.5). Many previous results for error correction schemes capable of handling deletions assumed a simple stationary uniform random error distribution.

We develop two different schemes: one for channels with deletions due to overt packet loss and one for channels without deletions. No deletions occur if there is no packet loss or if packet loss can be detected by the receiver, for example by using TCP sequence numbers. Our schemes are not limited to the TTL channel. As we show in Chapter 4, they can be used for other noisy covert channels as well.

3.4.1 Channel coding techniques

In general there are two types of techniques available for providing reliable data transport. In Forward Error Correction (FEC) schemes the sender adds redundancy that is used by the receiver to detect and correct errors. In Automatic Repeat Request (ARQ) schemes the sender retransmits data that the receiver has not received correctly previously. ARQ schemes require bidirectional communication since the receiver has to inform the sender about the corrupted or lost data. Hybrid approaches are also possible.

ARQ schemes require a sequence number and a checksum for each data block, so that the receiver can detect corrupted or lost blocks and inform the sender. FEC schemes add error correction information to each block. If the FEC decoder can determine reliably if a block has been decoded correctly an additional checksum is not needed, but sequence numbers are still required to identify undecodable blocks.

The efficiency of different techniques can be compared based on the *code rate*, which states the fraction of the transmitted payload data that is non-redundant. The code rate of a selective repeat ARQ scheme is:

$$\frac{(N-H)}{N}\frac{1}{\overline{T}\left(p_{\mathrm{E}},\hat{p}_{\mathrm{B}},N\right)} , \tag{3.16}$$

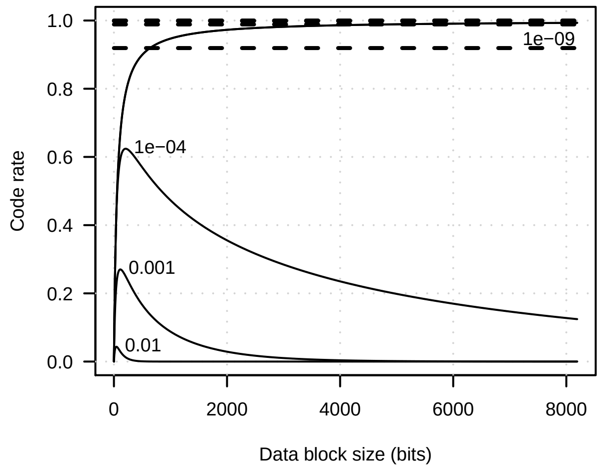


Figure 3.11: Code rates for ARQ schemes for different data block sizes and bit error rates

where N is the block size in bits, H is the number of header bits and $\overline{T}(p_{E,}\hat{p}_B, N)$ is the average number of (re)transmissions depending on N, the overall bit error rate p_E and a target block corruption rate $\hat{p}_B$ (see Appendix B.6).

Figure 3.11 depicts the code rates for a very low bit error rate (e.g. wired communications) and typical bit error rates of the TTL covert channel without overt packet loss and reordering, depending on the data size $(N-H)$ for a target block corruption rate of 1% and $H = 56$ (sequence number, checksum and corrupted block indicator). The dashed lines show the theoretical channel capacities for the same bit error rates.

For very low bit error rates and larger block sizes ARQ code rates are close to the channel capacity. However, for higher error rates the code rates are significantly reduced. For example, for typical error rates on the TTL channel between 1^{-4} and 1^{-3} the code rate reduces to between 0.6 and 0.3 although the channel capacity is close to 0.99. Therefore, FEC needs to be employed as it reduces the block corruption rate to acceptable levels with higher code rates.

However, since the channel errors are bursty (see Section 3.5.4), it is not very efficient to reduce the block error rate to zero with FEC alone. We construct FEC-based schemes on top of which standard ARQ mechanisms can be used.

3.4.2 Non-deletion channels

For non-deletion channels we use an existing error correcting code. Similar to the Cyclic Redundancy Check (CRC) based framing of Asynchronous Transfer Mode (ATM), we also use the code for identifying blocks in the received data.

We decided to use Reed-Solomon (RS) codes because they are well understood and fast implementations are readily available. RS codes are widely used, for example by

Digital Subscriber Line (DSL) and WiMax as well as on CDs, DVDs, and blue-ray discs. Furthermore, RS codes are suitable for channels with bursty errors.

RS codes are block codes. A (N,K) RS code has blocks of N symbols, with $N-K$ RS symbols appended to every K payload symbols. The maximum N depends on the size of the symbols in bits M ($N \leq 2^M - 1$). An RS decoder can correct $2E+S \leq N-K$ errors where E are erasures (symbols with bit errors of known position) and S are substitutions (symbols with bit errors of unknown position).

The sender divides the covert data into blocks. Each block has a header with an 8-bit sequence number, which enables the receiver to identify blocks lost due to corruption. The header also contains a 32-bit CRC (CRC32) checksum computed over the header fields and data, because the RS decoder we use [177] is not able to reliably indicate if all errors were corrected in a received block. The RS encoder computes the error correction data over the sequence number, checksum and covert data, and appends it to the block.

The receiver decodes blocks from the received bit stream as follows. For every new bit received it checks if N symbols are in the buffer already. If that is the case it attempts to decode a block using the RS decoder, and computes the CRC32 checksum over the corrected header and covert data. If the checksum matches the sender's checksum the received block is valid. Otherwise the receiver removes the oldest bit from the buffer and waits for the next bit.

Our protocol does not require synchronisation at the start. Any blocks sent by Alice before Bob started receiving are obviously lost, but Bob will start receiving data once the first complete block has been received.

We chose CRC32 as checksum because it provides better or equal error detection than other existing 32-bit checksums [178]. At very high error rates CRC32 may be too weak, but we assume that typically our scheme is used with lower error rates. Otherwise better checksums could be used at the expense of more computational or header overhead.

3.4.3 Deletion channels

A simple error-correction code is insufficient for channels with deletions because every deletion causes possible substitution errors in all following bits. Thus a decoder first has to identify where the deletions occurred and insert dummy bits. Then an existing error correcting code can be used to correct the errors caused by substitutions and dummy bits.

Ratzer developed an encoding scheme based on marker codes and Low Density Parity Check (LDPC) codes [179]. Marker codes insert sequences of known bits, so-called markers, at regular positions into the stream of payload bits. In Ratzer's scheme the inner marker code is used for re-synchronisation and the remaining substitution errors are then corrected by the outer LDPC code. He proposed probabilistic re-synchronisation (also referred to as sum-product algorithm).

Algorithm 3.1 Outer marker code receiver algorithm

```
function receive(bits)
  foreach bit in bits do
    recv_buffer = append(recv_buffer, bit)
    if sizeof(recv_buffer) ≥ sizeof(PREAMBLE) + MAX_OFFSET then
       foreach offset in 1...MAX_OFFSET do
         diff[offset] = difference(recv_block[offset], PREAMBLE)

       min_diff = min(diff)
       min_off = minarg(diff)
       if min_off = 0 and min_diff ≤ MAX_ERRORS and
           sizeof(recv_buffer) ≥ (EXPECTED_BLOCK_SIZE − Δ) then
          // return data block before preamble (if any)
          // remove bits from recv_buffer and continue
```

Ratzer's approach assumes that sender and receiver are synchronised at the beginning and approximate bit error rates are known to carry out the probabilistic re-synchronisation. The complexity of the algorithm is N^2 in time and space where N is the number of bits per block, although in practice one can often further limit the search space.

Because of these limitations we developed a scheme that follows a slightly different approach. Our scheme does not necessitate an initial synchronisation of sender and receiver (not always practical) or the prior knowledge of bit error rates (generally unknown). Furthermore, the complexity of our algorithm is only M^2 in time and M in space where M is the number of markers per block (usually $M \ll N$).

Our novel scheme is a hierarchical marker scheme that uses two layers of markers. The sender divides the covert data into blocks. An outer marker, which acts as preamble, is sent prior to each block. The receiver detects the start of a block if the bit sequence received is similar to the preamble. The preamble must be long enough to differentiate between preambles with bit errors and block data. The sender algorithm is trivial and consists only of putting the preamble at the start of each block. Algorithm 3.1 shows the receiver algorithm for the outer marker code.

The receiver appends every received bit to a temporary buffer. When there are enough bits in the buffer it computes the number of bits that differ between the received bit sequence and the pre-defined preamble for a number of offsets. If a bit sequence is detected with a difference smaller than a pre-defined threshold, differences for larger offsets are higher and the total number of bits received so far is over a minimum expected size the receiver assumes a preamble has been found.

The outer marker code enables the receiver to synchronise the start of blocks. The receiver then also knows approximately how many deletions occurred in a block including the preamble. To detect the locations of the deletions inside the block an inner marker code is used. If very few preamble bits are incorrectly identified as data bits the receiver underestimates the true number of deletions. Hence the inner marker code receiver algorithm is executed multiple times with increasing number of assumed deletions[1].

Ratzer showed that deterministic markers perform equal or better than random markers for channels with insertion/deletion rates of less than 1% [180]. We use deterministic markers as in his work. A marker is a number of zeros followed by a one, for example “001”. The sender algorithm uses an RS encoder to compute error correction data over header and payload, and then inserts an m bit marker after each n bits.

The receiver first tries to identify the positions of the deletions. The algorithm performs a search for a plausible marker sequence with the known number of deletions in the block as constraint (see Algorithm 3.2). The bits at every assumed marker position are checked and the number of missing bits is computed. For example, if the used marker is “001” and the bits checked are “010” then one bit has been deleted between the last and the current marker. Since marker bits can be corrupted themselves the number of deleted bits computed is possibly incorrect. Hence for each marker the algorithm also computes the number of possible bit errors. For example, if the assumed marker is “010” the bit preceding this sequence must be a zero otherwise a marker bit was deleted or substituted.

The variable `offset` keeps track of the total number of deletions identified so far. As long as the number of errors (`total_errors`) is below a threshold (`error_threshold`) the number of deletions for the current marker (`deletions`) is set as indicated by the assumed marker. However, since there cannot be more deletions than indicated by the outer marker code, `deletions` and the error are adjusted if the maximum is exceeded. If the error exceeds the threshold the algorithm backtracks assuming that mistakes were made. It backtracks to the previous marker with at least one assumed deletion, subtracts one and then resumes the forward search.

How aggressively the algorithm backtracks depends on `error_threshold`. The search is executed multiple times with `error_threshold` varying from zero to the maximum value `MAX_ERROR_THRESHOLD`. We found the choice of `MAX_ERROR_THRESHOLD` is not very critical for deletion rates of 1% or less, as long as it is not chosen too small[2]. After each search the receiver attempts to decode the block, unless the search produced the same solution as before or the algorithm failed to converge.

Dummy bits are inserted at the identified positions of deletions and the RS code is used to correct the dummy bits and other substitution errors (see Algorithm 3.3). RS codes correct errors on a per-symbol basis. This means when inserting dummy bits it does not matter where inside a symbol they are inserted. If the space between markers is only one symbol the RS decoder can correct the maximum $N-K$ symbols. However, if the space between markers is multiple symbols it is unknown in which symbol(s) the

[1]In the experiments we assumed a maximum of two such bit insertions.

[2]We set `MAX_ERROR_THRESHOLD`=5 in all experiments.

Algorithm 3.2 Inner marker code receiver algorithm

```
function receive(block)
  offset = 0
  deletion_list[1...markers] = 0
  last_deletion_list[1...markers] = 0
  error_list[1...markers] = 0
  assumed_deletions = EXPECTED_BLOCK_SIZE − sizeof(block)

  foreach error_threshold in 0...MAX_ERROR_THRESHOLD do

    foreach pos in 1...markers do
      index = (MARKER_SPACE + sizeof(MARKER))·pos + MARKER_SPACE − offset
      deletions = check_marker(index)
      if deletions > assumed_deletions − offset then
        deletions = 0
      error_list[pos] = compute_errors(index)

      foreach i in 1...pos do
        total_errors = total_errors + error_list[i]

      if total_errors < error_threshold then
        deletion_list[pos] = deletions
        offset = offset + deletions
      else
        pos = max(0, pos − 1)
        while pos > 0 and deletion_list[pos] = 0 do
          pos = pos − 1
        if deletion_list[pos] > 0 then
          deletion_list[pos] = deletion_list[pos] − 1
          offset = offset − 1
        index = (MARKER_SPACE + sizeof(MARKER))·pos + MARKER_SPACE − offset
        error_list[pos] = compute_errors(index)
    if assumed_deletions > offset then
      deletion_list[markers] = assumed_deletions − offset

    has_converged = check_convergence(deletion_list)
    if has_converged and deletion_list ≠ last_deletion_list then
      decode_block(block, deletion_list)
    last_deletion_list = deletion_list
```

deletion(s) occurred and the RS decoder can only correct $\frac{N-K}{2}$ errors. There is a trade-off between the overhead of the inner marker code and the RS code.

Since covert channels have a low bit rate we also explore another trade-off between code rate and decoding time. If the number of possible combinations of symbols with deletions is reasonably small the decoder can attempt to decode the received block for each possible combination. For example, if the symbol size is 8 bits and markers are spaced 16 bits apart the receiver needs to search through 2^{M_D} combinations where M_D is the number of markers with deletions. For each combination the receiver executes the RS

Algorithm 3.3 Block decoding algorithm

```
function decode_block(block, deletion_list)
  foreach pos in 1...markers do
    if deletion_list[pos] > 0 then
      block = insert(block, pos, deletion_list[pos])

  corrected = rs_decode(block, deletion_list)
  send_crc = get_sender_crc(block)
  recv_crc = compute_crc(block)
  if corrected ≥ assumed_deletions and recv_crc = send_crc then
    // return valid block
```

decoder[3]. A valid block is found if the RS decoder is able to correct all errors and the block checksum is valid.

This ‘brute-force’ decoding increases the code rate and is feasible for a small number of symbols between markers and small deletion rates. Also, if the redundancy of the RS code is much higher than needed on average a solution is found well before all combinations have been tried. Our later analysis shows that the average number of bits decoded per second is still much higher than the maximum transmission rate.

3.5 Empirical evaluation

First we analyse the error rate of the different modulation schemes without reliable transport. We emulate the covert channel using overt traffic from different trace files and measure the resulting bit error rates. Based on the channel model presented in Section 3.3 we then compute the channel capacities and transmission rates, and compare the different modulation schemes. We also investigate the burstiness of the errors.

Later we evaluate the throughput of the channel using the techniques for reliable transport described in the previous section. We analyse the throughput achieved for large aggregates of overt traffic taken from traces as well as for single overt flows generated by specific applications in a testbed. We compare the throughput with the channel capacity.

3.5.1 Datasets and methodology

The Covert Channels Evaluation Framework (CCHEF), described in Appendix A, can emulate the use of covert channels based on overt traffic from traces. This makes it possible to evaluate the TTL covert channel with large realistic traffic aggregates that are impossible to create in a testbed. The overt traffic was taken from the traces described in

[3]The maximum number of tried combinations is limited to a configurable value.

Section 3.1. For performance reasons we did not use the full traces, but selected representative pieces. We used 24 hours of the CAIA, Grangenet, Bell, Twente and Waikato traces, and six hours of the Leipzig trace (each containing between 20 and 261 million packets). We used traffic in both directions.

In a first pass we computed the TTL error for each packet in each trace as described in Section 3.1.5. Then during the actual covert channel emulation the TTL field was modified according to the pre-computed TTL error after the covert channel modulation. Alice and Bob used all overt packets available for the covert channel. Alice sent uniform random covert data, as if the data had been encrypted. Because of the random input data we repeated each experiment three times and report the mean error rate. Since the standard deviations are very small we do not include errors bars in the graphs.

Let A be the amplitude of the modulation schemes (difference between the signal level of logical zero and one). For direct schemes $A = 1$, for DUB $A = 2\Delta$ and for all other techniques $A = \Delta$. From Section 3.1 it is clear that only one bit per packet (direct, mapped schemes) or packet pair (differential schemes) can be encoded and A must be minimal, as otherwise the channel would not look like normal TTL variation[4]. Hence in our experiments we used binary channels and the TTL amplitude was only varied within a narrow range ($1 \leq A \leq 6$) to investigate its influence on the error rate.

For direct encoding schemes we assumed knowledge of the true hop count at the receiver. For mapped schemes we considered the cases where the receiver 1) knows the mapping and 2) learns the mapping from the extra zero bit (see Section 3.2).

Alice can only modify the TTL field to values between a minimum value and the maximum value of 255. She must avoid setting small TTL values that could result in packets being discarded on the way to the receiver. This limitation inevitably causes bit errors for schemes that increase TTL and tend to wrap-around (MEI and DUB). For schemes that decrease TTL bit errors are unlikely for small A as initial TTL values are usually at least 64 (see Section 3.2.2).

In our experiments we leveraged the fact that the trace files were captured on end hosts or access routers to select the minimum TTL value. If a packet traversed ≤ 4 hops the minimum TTL was set to 31 minus the hops already traversed (outgoing packet). If a packet traversed > 4 hops the minimum TTL value was set to 4 (incoming packet). In practice other strategies could be used, for example Alice could select the minimum TTL for each packet based on its destination address.

[4]If used as side channel, encoding multiple bits with larger amplitudes is perfectly reasonable.

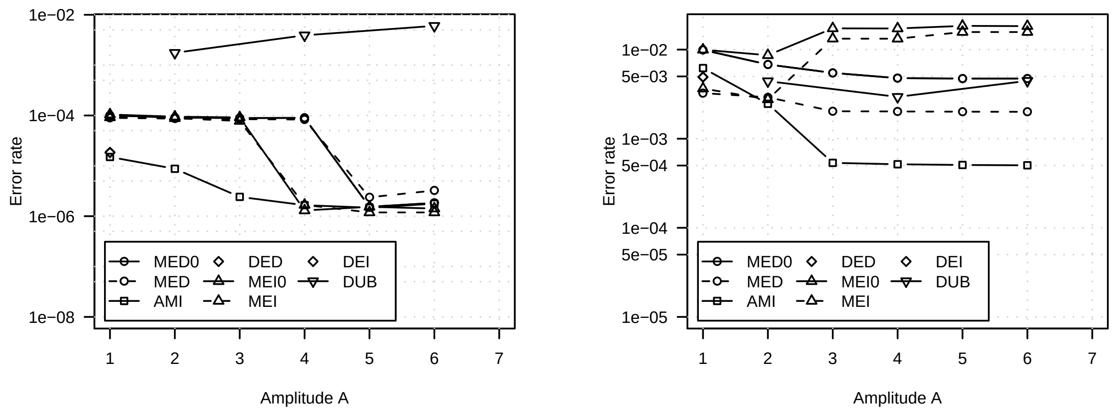


Figure 3.12: Error rate for different modulation schemes and amplitudes for the CAIA trace (left) and the Leipzig trace (right) (y-axis is logarithmic)

3.5.2 Error rate

Figure 3.12 shows the error rate for the CAIA and Leipzig datasets depending on the different modulation schemes and A (results for other traces are in Appendix B.3). Overall the error rates for $A \leq 2$ are similar to the average rate of TTL changes (see Section 3.1). A notable exception is DUB for the CAIA trace. Since this trace contains a large number of very long flows the probability for TTL wrap-arounds is much higher than for the other traces. The error rate for MEI and DUB actually increases for increasing A because the probability of wrap-arounds increases.

MEI0 and MED0 have higher error rates than MEI and MED. This is because often the probability that the first special zero bit is wrong is higher than the average error rate, since TTL changes occur more frequently at the start of flows (see Section 3.1). Both direct schemes and mapped schemes perform equally well for $A = 1$ as predicted by the error probabilities (see Appendix B.2). In general the error rate does not decrease proportionally with increasing A because the empirical error distributions have long tails as shown in Section 3.1.5.

Figure 3.13 compares the performance of the different modulation schemes averaged across all traces. For $A = 1$ the error rate varies between 0.1% and 1%, and MED performs best, followed by MEI, AMI and the direct schemes. For larger amplitudes MED and AMI outperform all other schemes. MEI and MED clearly outperform MEI0 and MED0.

We investigated if the error rate for mapped and differential schemes can be reduced by using hop count differences instead of TTL differences. The receiver converts all TTL values to hop counts. This eliminates errors when the TTL was changed by middleboxes but the hop count is the same (see Section 3.1.3). For example, the TTLs 56 and 120 are different but the hop count is 8 in both cases assuming the usual initial TTL values. Our

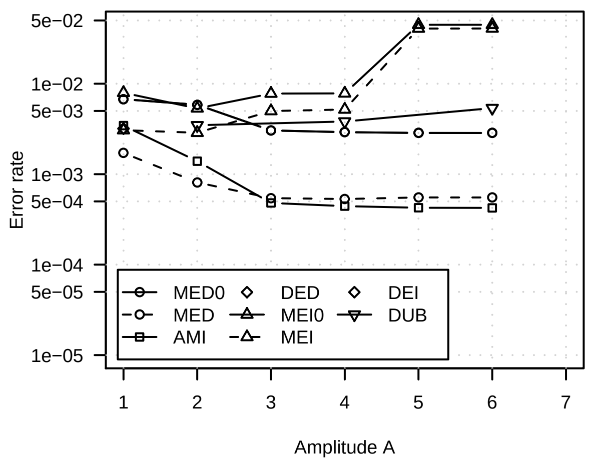


Figure 3.13: Average error rate of different modulation schemes across all traces

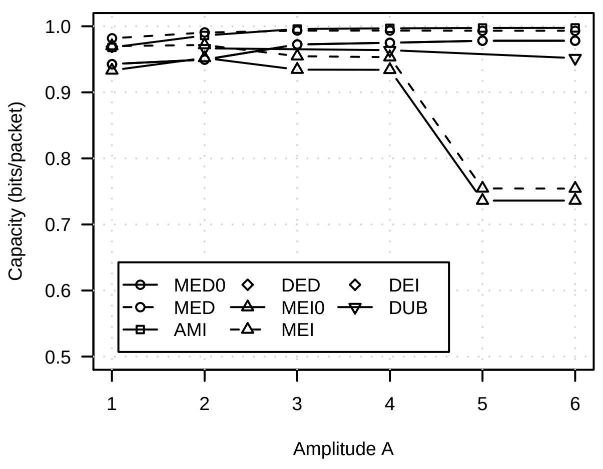


Figure 3.14: Capacity in bits per overt packet or packet pair for all modulation schemes based on the average error rate (without packet loss or reordering)

results indicate that there is little benefit for small amplitudes, but for larger amplitudes there is some improvement (see Appendix B.4).

3.5.3 Capacity and transmission rate

Knowing the different error rates we now estimate the channel capacities and maximum transmission rates. First we analyse the case without packet loss or reordering and compute the capacity using Equation 3.12. Figure 3.14 shows the capacity in bits per overt packet (direct, mapped) or packet pair (differential) for the different modulation schemes based on the average error rates over all traces.

Table 3.5: Average transmission rates in (kilo) bits/second

Dataset	Direct	MEI	MEI0	MED	MED0	DUB	AMI
CAIA	71(.99)	71(.99)	65(.99)	71(.99)	65(.99)	64(.98)	65(.99)
Grangenet	68(.99)	68(.99)	59(.94)	68(.99)	59(.94)	61(.98)	62(.99)
Twente	482(.97)	483(.99)	438(.98)	483(.99)	438(.98)	444(.99)	445(.99)
Waikato	1397(.92)	1399(.97)	1096(.89)	1423(.99)	1102(.89)	1200(.97)	1204(.98)
Bell	220(.89)	219(.91)	204(.89)	229(.95)	213(.94)	210(.92)	207(.91)
Leipzig	11.6k(.96)	10.7k(.97)	10.2k(.92)	11.7k(.97)	10.2k(.92)	10.6k(.96)	10.5k(.95)

The transmission rates depend on the capacity in bits per overt packet and the average packet rates (Equation 3.14). Table 3.5 shows the average transmission rates in bits per second for all modulation schemes and traces assuming $\Delta = 1$, no packet loss/reordering and hop count differences are used for MED, MED0 and AMI schemes[5]. The numbers in parenthesis denote the capacity in bits per overt packet.

Overall the transmission rates depend on the available overt traffic, varying from tens of bits per second (CAIA, Grangenet), over hundreds of bits per second (Bell, Twente), to up to several kilobit per second (Leipzig, Waikato). Besides being standards-compliant and stealthier our novel schemes (MED, AMI) also have equal or higher transmission rates than the previous schemes (MEI, DUB). Overall we rank the new schemes as follows (from best to worst): MED, DED, AMI, MED0.

With increasing packet loss and reordering rate the channel capacity reduces quickly. Figure 3.15 shows a contour plot of the capacity depending on the loss and reordering rate if packet loss can be detected (Equation 4.10). Figure 3.16 shows the capacity when packet loss cannot be detected (Equation 4.9). In both figures p_N is the average error rate for MED with $A = 1$. For reordering we set $b = 1$, $d = 1$, and $r = 1/\text{rate} - 2$ (see Section 3.3), where rate is the reordering rate shown on the x-axis.

3.5.4 Burstiness of errors

The burstiness of errors does not affect the channel capacity, but it affects the performance of techniques for reliable data transport. On the TTL channel bit errors often occur in bursts. How bursty the errors are depends on the trace and the modulation scheme.

We illustrate this using results for the MED and AMI modulation schemes and the CAIA and Leipzig traces. Figure 3.17 shows CDFs of the distance between errors in bits for the measured errors and simulated uniformly distributed errors with the same probabilities. The empirical error distributions are clearly burstier than the uniform distributions.

[5]The transmission rates can be increased by encoding multiple bits per packet or packet pair. However, as explained earlier this would likely reveal the covert channel.

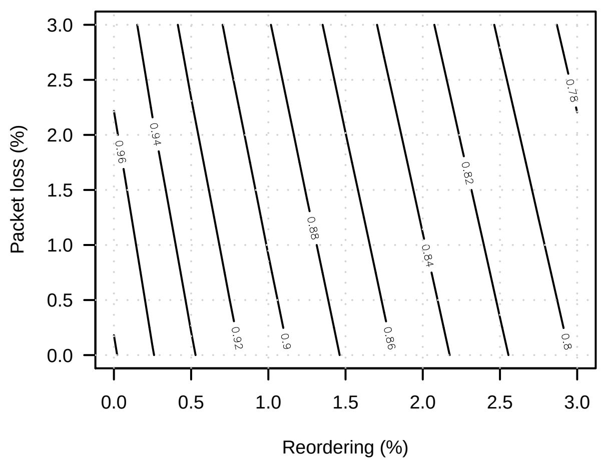


Figure 3.15: Capacity depending on packet loss and reordering if loss can be detected

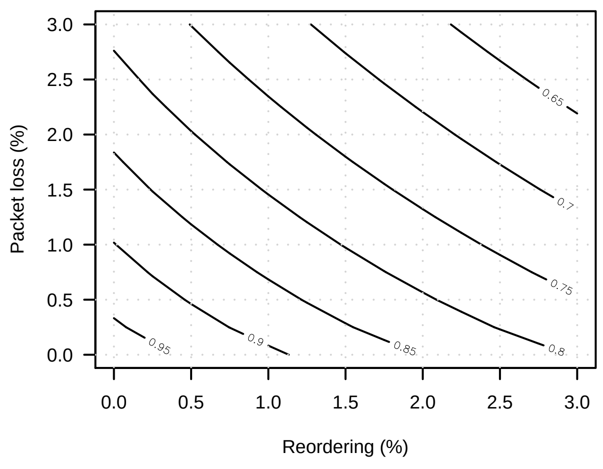


Figure 3.16: Capacity depending on packet loss and reordering if loss cannot be detected

Figure 3.18 plots the error rate for the MED scheme for consecutive non-overlapping windows of 100 000 bits (equivalent to 100 000 overt packets) for the two largest traces (Waikato, Leipzig). There is high short-term variation for the Leipzig trace. The Waikato trace shows smaller short-term variation combined with small long-term changes. For the other traces the error rate is also changing, although usually not as dramatically as in the Leipzig trace (see Appendix B.5).

Larger changes of the error rate mean a non-adaptive error correction code would either have too much overhead most of the time or the remaining error rate is higher than in the case of uniformly distributed errors. Optimally, the sender should adapt the error correction code over time based on the estimated error rate. For bidirectional channels the error rate estimate could be based on feedback from the receiver.

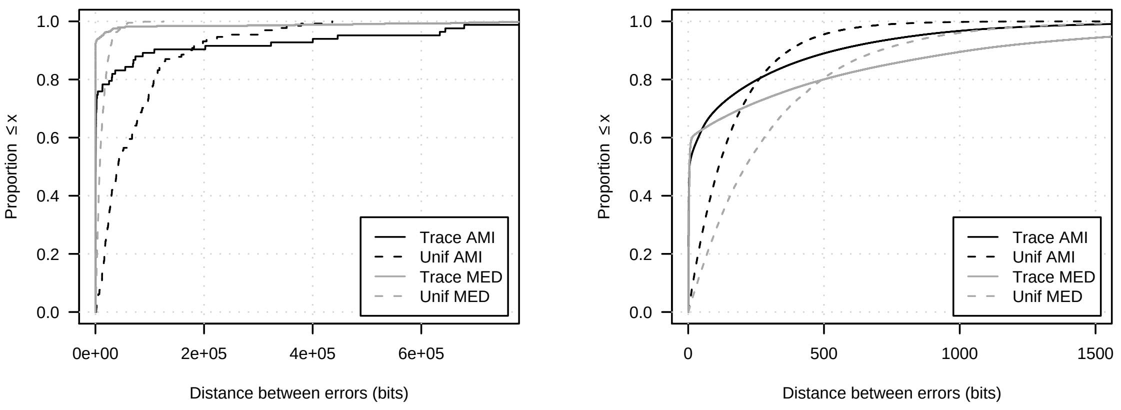


Figure 3.17: Distance between bit errors in bits for the CAIA trace (left) and the Leipzig trace (right)

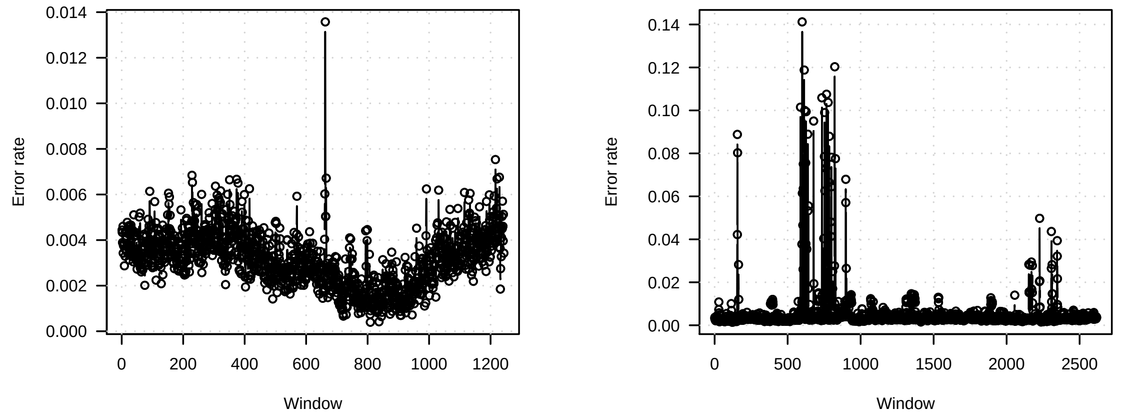


Figure 3.18: Error rate for the MED modulation scheme over consecutive windows of 100 000 bits for the Waikato trace (left) and the Leipzig trace (right)

3.5.5 Throughput – trace file analysis

We measured the throughput of the channel using the proposed techniques for reliable transport and large traffic aggregates with real TTL noise as overt traffic. We simulated packet loss and reordering since it cannot be deduced from the trace files. We only used the DED, MED, MED0 and AMI encoding schemes because they perform equal or better than the other schemes.

Previous studies showed that although some paths experience high loss rates, packet loss in the Internet is typically $\leq 1\%$ over wired paths [181, 182, 183, 184]. Previous studies on packet reordering found that it is largely flow and path dependent [182, 185, 186]. For example, Iannacone *et al.* found that while 1–2% of all packets in the Sprint backbone were reordered, only 5% of the flows experienced reordering [185].

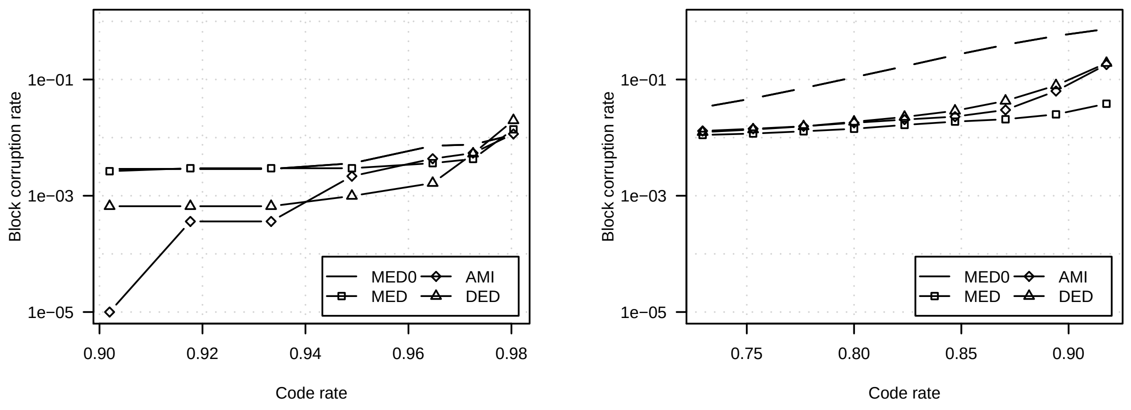


Figure 3.19: Block corruption rate depending on code rate for CAIA (left) and Leipzig (right) for 0% packet loss and reordering

We used packet loss rates of 0%, 0.1%, 0.5% and 1% with uniform distribution. For packet reordering we used rates of 0%, 0.1% and 0.5% using a uniform distribution, and each reordered packet was swapped with the previous packet ($d = 1$ and $b = 1$, see Section 3.3). We limited our analysis to a small set of loss and reordering rates to keep the effort reasonable. Also, for high rates the capacity is expected to be low.

For the RS code we always used one-byte symbols, limiting the maximum block size to 255 bytes. We used the largest possible block size for the non-deletion technique and varied sizes for the marker-based technique depending on the packet loss and reordering rate (see Appendix B.11). Longer RS codes could be used with larger symbols, but then the decoding time and the time for transmitting blocks increase. Since the rate of the covert channel is low it takes a while to transmit a data block. For example, for the CAIA trace on average it takes around 30 seconds to transmit one 255 byte block.

The following graphs show the average rate of corrupted blocks over increasing code rate. We only show some of the graphs for the CAIA and Leipzig traces (graphs for other traces are in Appendix B.7). Figures 3.19 and 3.20 show the block corruption rates for 0% and 0.5% packet loss with 0% packet reordering.

Figure 3.19 shows that for the CAIA trace AMI and DED outperform both MED schemes, but for Leipzig the performance of most schemes is similar. MED0 performs by far the worst across all traces because of the higher bit error rates, as discussed in Section 3.5.2. In the rest of the experiments we omitted MED0 since the high error rates result in extremely long analysis times.

Figure 3.20 shows that AMI performs clearly worse than MED and DED. Packet loss causes additional substitution errors for differential schemes, such as AMI, as discussed in Section 3.3. MED and DED perform equally good, with MED being better at high code

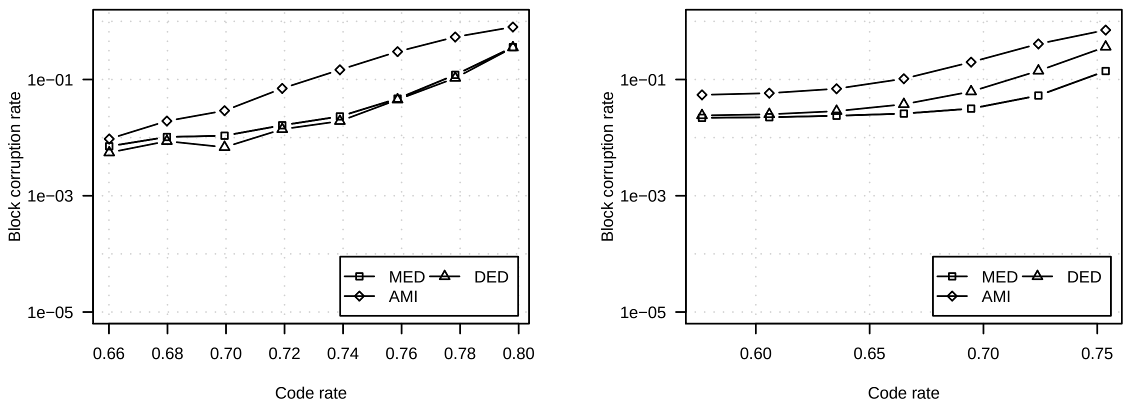


Figure 3.20: Block corruption rate depending on code rate for CAIA (left) and Leipzig (right) for 0.5% packet loss and 0% packet reordering

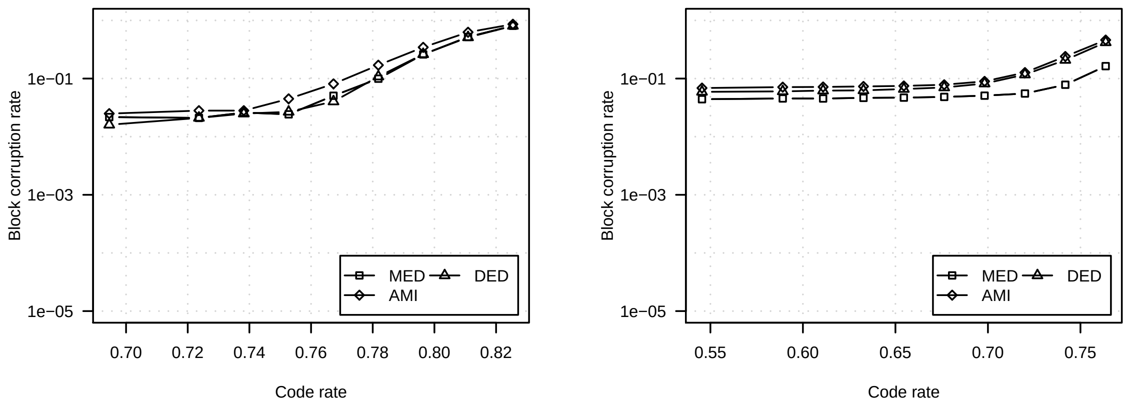


Figure 3.21: Block corruption rate depending on code rate for CAIA (left) and Leipzig (right) for 0.1% packet loss and 0.5% packet reordering

rates for Leipzig. In the experiments with 1% packet loss we omitted AMI because of the long analysis times.

Figure 3.21 shows the block corruption rate for 0.1% packet loss and 0.5% packet reordering. The block corruption rate is worse than for 0.5% packet loss without reordering. In general reordering is worse than loss for our reliable transport technique. For each reordering there is a 50% chance of two wrong consecutive bits. Even if these fall into one data symbol two RS symbols are needed for correction compared to only one RS symbol needed per erasure. In the worst case the bits are in two different symbols and four RS symbols are required for correction.

In all the figures we see that at some point the block corruption rate does not decrease quickly anymore with increasing amount of redundancy. This is because of the burstiness of the errors. Few large bursts require a much larger amount of redundancy than needed

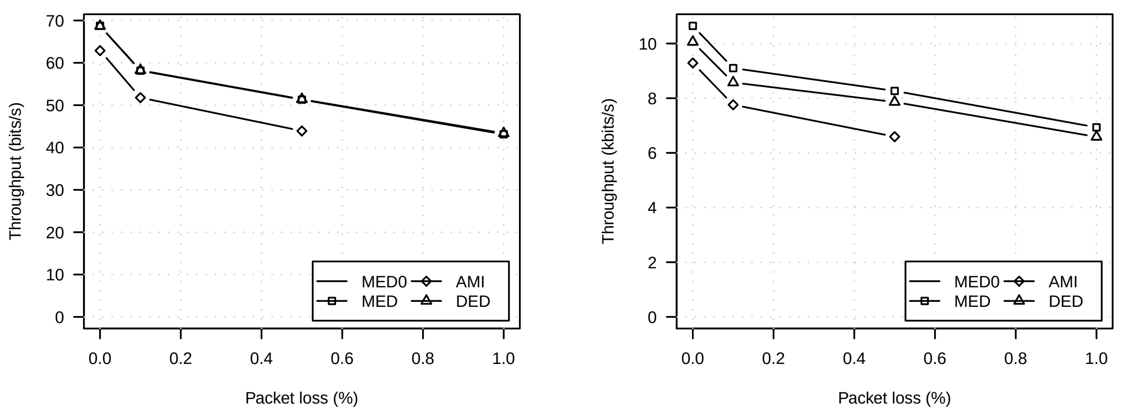


Figure 3.22: Throughput depending on packet loss for CAIA (left) and Leipzig (right) for 0% packet reordering

on average. However, a reasonably large amount is able to reduce block corruption rates to levels where ARQ is effective.

Based on the block size N and the remaining block corruption probability after FEC $\tilde{p}_B$ achieved with code rate R_{FEC} we computed the average number of retransmissions $\overline{T}$ needed to reliably transmit all remaining corrupted blocks (see Appendix B.6). We assumed selective repeat ARQ is used with H_{ARQ} bits informing the sender which blocks need to be retransmitted. Then the resulting code rate of a hybrid FEC+ARQ scheme is:

$$R_{FEC\_ARQ} = \left[(1-\tilde{p}_B) + \frac{\tilde{p}_B}{\overline{T}(\tilde{p}_B, \hat{p}_B)}\right] R_{FEC} \frac{N - H_{ARQ}}{N} . \quad (3.17)$$

Assuming a target block corruption rate $\hat{p}_B = 1^{-9}$ and $H_{ARQ} = 16$ bits (each block contains two additional 8-bit numbers indicating the start and end sequence numbers of bursts of corrupted blocks) we computed the best code rates as the maximum code rate from Equation 3.17 over all empirically measured rates R_{FEC}.

Figure 3.22 shows the throughput for the CAIA and Leipzig traces depending on packet loss without packet reordering. Figure 3.23 shows the throughput for the same traces depending on packet reordering for a packet loss rate of 0.1% (results for other traces are in Appendix B.8). The hybrid scheme works well as the decrease in throughput is only moderate for increasing packet loss and reordering. The best performing schemes are MED and DED.

Figure 3.24 shows the percentage of the channel capacity reached for the different modulation schemes averaged across all traces. Overall MED performs best. DED also performs well, especially for traces that contain a large percentage of long UDP flows. AMI performs well if there is no packet loss. Overall, the FEC+ARQ transport protocol achieves at least 60% of the capacity.

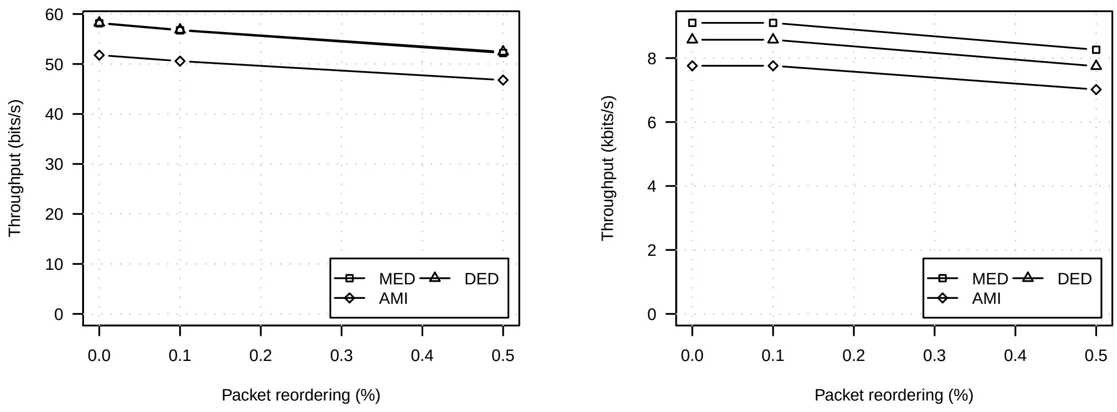


Figure 3.23: Throughput depending on packet reordering for CAIA (left) and Leipzig (right) for 0.1% packet loss

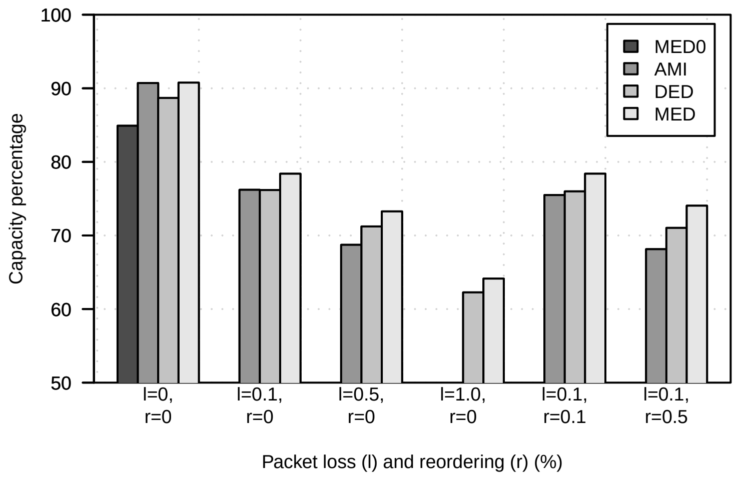


Figure 3.24: Percentage of capacity reached for the different modulation schemes and the different packet loss and reordering settings averaged across all traces

We also measured the average block decoding speed of the receiver, which is defined as the length of a data block in bytes divided by the average decoding time of blocks. The differences between different modulation schemes are negligible, but there are significant differences between the different traces. The results show that even for the experiments with the highest error rate the decoder is still much faster than the actual throughput of the channel (see Appendix B.9).

3.5.6 Throughput – testbed experiments

In the testbed experiments we measured the throughput of the covert channel across a real network for different types of overt traffic. The covert channel was encoded in a single overt traffic flow. This setup is representative for a scenario where the overt traffic is limited to single flows. Our testbed consisted of two computers[6] connected via a Fast Ethernet switch. Alice and Bob were co-located with the actual sender and receiver, but they could have been middlemen.

We used different applications with different packet rates as overt traffic. We used scp to perform file transfers capped at 2 Mbit/s, and SSH to perform remote interactive shell sessions (recorded and repeatedly played back with XMacro [187]). We generated game traffic using Quake III Arena (Q3) [188], with bots as players. These applications represent bulk transfer, interactive and real-time traffic classes.

Each experiment lasting 20 minutes was repeated three times and we report the average throughput. We emulated packet loss and reordering rates of 0%, 0.1%, 0.5% and 1% using Linux Netem [189]. For both we configured a correlation of 25% [189], because in the Internet packet loss and reordering are typically bursty. The delay was set to fixed 25 ms in each direction, except for the experiments with reordering.

With Netem's packet reordering the resulting bit error rate depends not only on the configured reordering rate, but also on the configured delay and the traffic's inter-packet times [189]. We used different delays for the different applications to achieve similar error rates, but scp still experienced slightly higher error rates than Q3 and SSH. We verified the accuracy of Netem prior to performing the experiments (see Appendix F).

As we showed in Section 3.1 TTL noise basically falls into two categories: deterministic and random. Since deterministic noise generally only occurs at the start and end of flows it would not have much impact here given the long duration of the flows. Hence we used CCHEF to emulate uniformly random TTL noise with Normal-distributed amplitudes using error rates of 0%, 0.001%, 0.1% and 1%.

We only used the MED modulation scheme but limited tests indicated a similar performance of the DED scheme. Without packet loss and reordering the non-deletion technique was used to provide reliable transport. Otherwise the marker-based technique was used. For simplicity we used the same codes for the different applications. In the experiments with packet reordering the codes were dimensioned for scp, and hence they were slightly less efficient for Q3 and SSH. The code parameters are given in Appendix B.11.

The encoding parameters were manually tuned according to the error rates in the different scenarios with the goal of achieving a very low block corruption rate with FEC alone. As before we computed the actual throughput assuming a FEC+ARQ scheme with a target block corruption rate of 1^{-9}. Because we used relatively high redundancies, in all experiments the block corruption rate was below 0.5% with FEC alone. In most experiments it was actually zero.

[6]An Intel Celeron 2.4 GHz with 256MB RAM and an Intel Celeron 3.0 GHz with 1GB RAM, both running Linux 2.6.18.

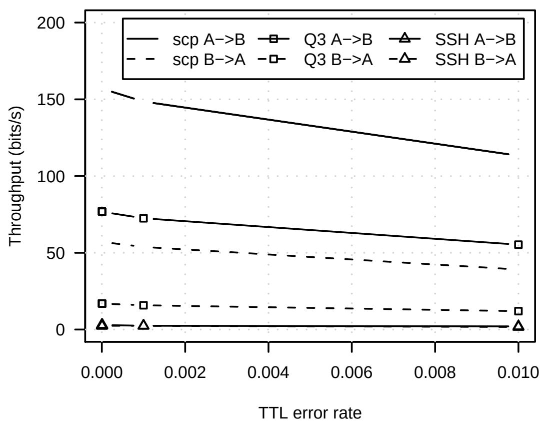


Figure 3.25: Throughput depending on the TTL error rate for 0% packet loss and reordering

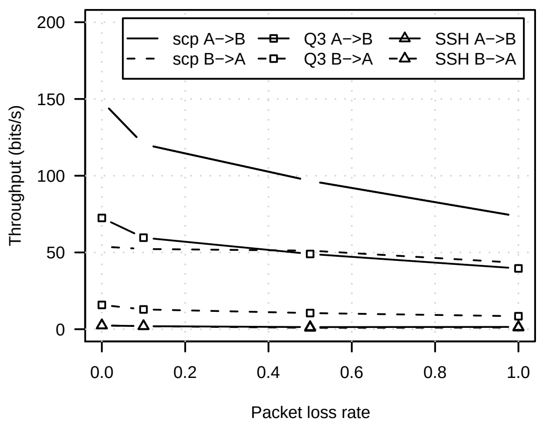


Figure 3.26: Throughput depending on the packet loss rate with 0.1% TTL error rate and 0% packet reordering

We used smaller code lengths for the testbed experiments to increase the total number of blocks, given that the packet rates were much smaller than the rates in the traces. Because of the smaller block sizes and the higher redundancies the code rates are smaller than in the trace-file analysis.

Figure 3.25, 3.26 and 3.27 show the throughput for the different applications and error rates. For scp the data was transferred from Alice to Bob and hence the rate of overt packets in that direction is higher. For Q3 the throughput from Alice to Bob is much larger, as Alice's host running the Q3 client sent one packet every 10–20 ms, but Bob's host running the server only sent one packet every 50 ms [190]. SSH throughput is roughly symmetric with the used shell commands.

Figure 3.28 shows the percentage of the channel capacity reached for the different applications averaged over both directions. The percentage should be roughly equal for all applications. However, SSH performs worse since the number of data blocks transmitted

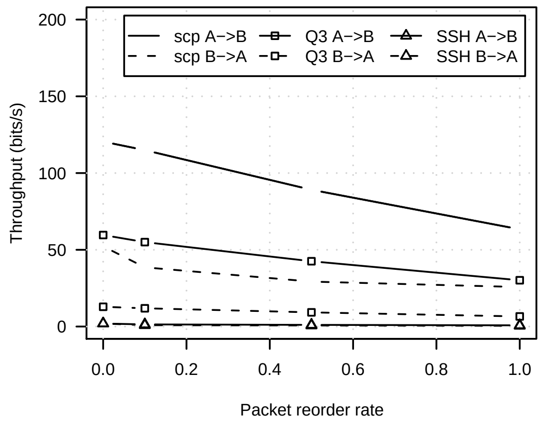


Figure 3.27: Throughput depending on the packet reordering rate with 0.1% TTL error rate and 0.1% packet loss rate

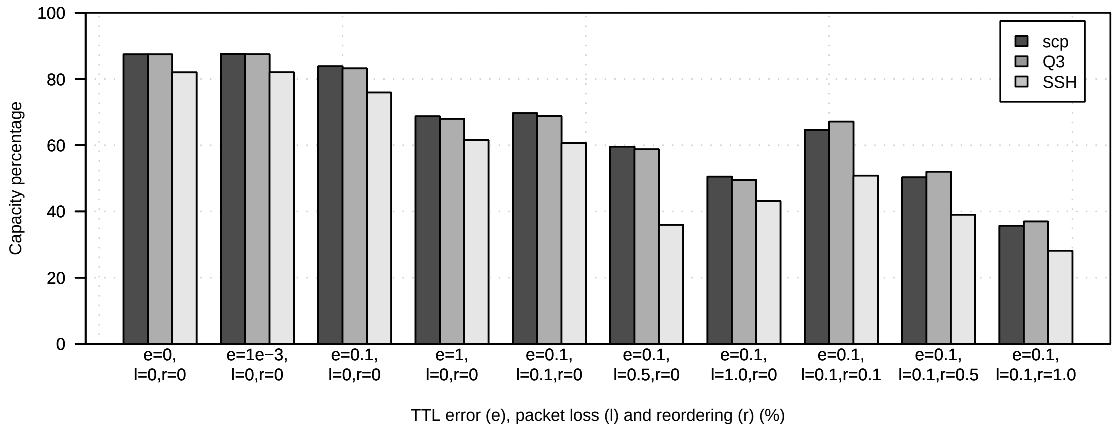


Figure 3.28: Percentage of capacity reached for the different applications and the different TTL error, packet loss and reordering rates averaged over both directions

is very small and incomplete blocks at the end of experiments do not count (rounding errors). Overall, the reliable transport protocol achieves at least 50% of the capacity, except in the case of high packet reordering.

Again, the results show that our reliable transport technique is more efficient for loss than reordering. The percentage of the capacity reached is 10–14% lower than in the trace file experiments due to the smaller code sizes and the higher redundancy. The smaller code size alone reduces the code rate by 5–6% given the overhead of the fixed header. The throughput is still in the order of tens of bits per second for applications with higher packet rates, such as scp or Q3 client-to-server traffic.

We also investigated the variability of the covert bit rate over time (see Appendix B.10). For Q3 and SSH it is almost constant, even with packet loss and reordering. For scp the rate is relatively constant without packet loss, but with packet loss it fluctuates.

If the overt traffic is TCP Alice and Bob could use TCP sequence numbers to mitigate the effects of packet loss and reordering. Alice has to store tuples of bits sent and TCP sequence numbers in a buffer. When she detects a TCP retransmission she must re-encode the bits sent previously. Bob needs to buffer received packets and put them in the right order, according to the TCP sequence numbers before decoding.

3.6 Conclusions

We analysed the characteristics of normal TTL variation from several traffic traces. We showed that normal TTL changes only occurred in less than 1% of packet pairs, but in 2–6% of the flows. The large majority of flows with changes had only two different TTL values differing by one. This noise reduces the channel capacity, but on the other hand it improves the stealth of the channel. Without normal TTL variation the covert channel would be trivial to detect.

We presented several novel improved modulation schemes. Our new schemes are stealthier and can be used with passive channels. Furthermore, they provide up to 5% higher capacities than previous schemes. However, even with the improved schemes the channel is still detectable, as we will show in Chapter 7.

We then proposed an information-theoretic model for the channel that can be used to determine the channel capacity based on errors caused by normal TTL variation, packet loss and packet reordering. The model is not limited to the TTL channel; it could be applied to direct storage channels in other IP header fields. We also developed techniques for reliable data transmission over the covert channel.

Since the TTL noise distributions are complex and cannot be modelled easily we analysed the error rates of the different modulation schemes by emulating the covert channel using overt traffic from traces. For a minimum TTL amplitude the average error rates across all traces vary between 1^{-3} and 1^{-2}. Larger amplitudes reduce the error rates but also reveal the covert channel, given the characteristics of normal TTL variation.

Based on the channel model and the measured error rates we estimated the capacities and transmission rates. Without packet loss and reordering the capacity is over 0.9 bits per overt packet or packet pair. But it reduces quickly with increasing packet loss and reordering. The transmission rates range from tens of bits per second up to a few kilobits per second.

We carried out several experiments to evaluate the throughput of the reliable transport technique. We emulated covert channels using overt traffic from traces and simulated packet loss and reordering. With a hybrid FEC+ARQ scheme we achieved throughputs of 60% or more of the capacity, with rates of up to several hundreds of bits per second.

We also conducted experiments over a real network using three different applications as overt traffic. For a hybrid FEC+ARQ scheme we achieved throughputs of 50% or more of the capacity, except in the case of high packet reordering. The throughput is up to over hundred bits per second, much higher than the commonly accepted covert channel limit of one bit per second [19].

3.6.1 Future work

The study of the channel characteristics could be extended towards further trace files. The capacity and throughput analysis could also be extended to cover a wider range of packet loss and reordering settings. Furthermore, experiments could be carried out across different Internet paths, for example using the PlanetLab overlay network.

Improved modulation schemes should be developed to make the TTL channel stealthier. Optimally covert sender and receiver would select the overt packets such that the distribution of the induced TTL changes looks exactly like normal TTL noise. The TTL noise distributions cannot be modelled easily with standard statistical distributions, but covert sender and receiver could use more complex models calibrated on observed traffic.

The performance of the technique for reliable data transport could be further improved. Longer RS codes would be more effective as header overhead is reduced, but then data is received in a less timely fashion. Although RS codes perform well, there are better error correcting codes, for example LDPC codes [179]. Furthermore, there may be other approaches that are more efficient than a hierarchical marker scheme.

Since the error rate of the TTL channel varies significantly over time depending on the overt traffic, it is questionable if a single error-correcting code could perform well in different circumstances. Developing and evaluating an adaptive scheme is left for further study. Another avenue left to explore is how much performance could be improved by reducing the burstiness of errors through interleaving of the data prior to encoding.

When encoding the covert channel into TCP flows, the effects of packet loss and reordering can be mitigated by utilising TCP sequence numbers. We outlined the design of such a scheme, but an implementation and evaluation are still missing.

Chapter 4

Packet-timing Covert Channels

In this chapter we analyse direct timing covert channels encoded in inter-packet times, also referred to as Inter Packets Gaps (IPGs). We chose channels in IPGs, because unlike packet rate channels they do not require synchronisation of time intervals [100] and they can be encoded into arbitrary IP traffic. Packet rate channels likely raise suspicion since many applications do not 'randomly' change packet rate over time.

Several encoding schemes were developed for IPG channels (see Section 2.2.5), but all the schemes prior to [104, 105] are easy to detect [107]. Gianvecchio *et al.* [104] and Sellke *et al.* [105] proposed channels in IPGs that are hard to detect because they perfectly mimic the shape of the IPG distributions of real applications. However, they destroy existing auto-correlations of IPGs. Thus they are hard to detect only if the IPGs of real applications are independent identically-distributed (iid) and hence uncorrelated.

Paxson *et al.* showed that Telnet traffic exhibits this behaviour [182]. Presumably IPGs of other human-driven application traffic, such as interactive SSH, are also iid. However, we show that IPGs of UDP-based game and VoIP traffic often exhibit moderate to strong correlation. Even TCP traffic has moderate correlation.

Previous work focused on TCP traffic, because it currently dominates the Internet [104, 105]. However, the percentage of UDP traffic is expected to become much larger in the future [191]. The channel proposed in [104, 105] requires accessible sequence numbers in the overt traffic; otherwise lost packets desynchronise covert sender and receiver. TCP provides sequence numbers, but not all UDP-based traffic has sequence numbers, or they may not be accessible if the protocol is encrypted.

Furthermore, previous research assumed that the overt traffic is generated by the covert sender [104, 105]. This means the channel cannot be used in scenarios where Alice is a middleman, and it leaves her with the burden of mimicking real traffic properly. If some characteristics of the 'fake' traffic are abnormal, the channel could be easily detected.

We propose new improved encoding techniques. Our improved channel maintains existing auto-correlations and thus is harder to detect even when IPGs are not iid (see Chapter 7). It can be used as active or as (semi-)passive channel. Our techniques generate the random numbers needed for encoding and decoding [104] from the packets themselves, which makes the channel robust enough for use with all UDP-based protocols.

First, we analyse IPGs in the traffic of different applications and show that often there is auto-correlation. We then present the improved covert channel. We develop one modulation technique that improves the stealth by reducing the capacity, and another technique that increases the stealth but has reduced robustness. We propose an information-theoretic channel model and derive the channel capacity.

We describe a proof-of-concept implementation and analyse its performance in a testbed, depending on network jitter, packet reordering and loss. We compute the capacity and maximum transmission rate based on the channel model and empirically measured error rates. The capacity is only 70–80% of the capacity of the TTL channel, but transmission rates are still up to over hundred bits per second. Finally, we measure the throughput across the channel using the reliable transport technique developed in Section 3.4 and find that the achieved throughput is at least 30–40% of the channel's capacity.

4.1 Inter-packet time analysis

We analyse two UDP-based applications, the First Person Shooter (FPS) game Quake III Arena (Q3) and the VoIP application Skype. We also analyse the IPGs of UDP and TCP flows taken from a traffic trace. We analyse the correlation of the series of IPGs as well as the series of least significant parts of IPGs. We define the least significant part as:

$$d_{\mathrm{lsp}} = d \bmod l = d - \left\lfloor \frac{d}{l} \right\rfloor l\,, \tag{4.1}$$

where d is the IPG and l is the size of the least significant part. For example, if the IPG is 21.75 ms and $l = 1$ ms then $d_{\mathrm{lsp}} = 0.75$ ms (sub-millisecond part).

Auto-correlation of IPGs tends to exist because of large-time-scale behaviours, such as the congestion window growth and collapse for TCP or the codec's encoding rates for VoIP over UDP, but the small time-scale behaviour exposed by looking only at the least significant part is jittered by largely uncorrelated noise, for example queuing delays at each hop. We demonstrate that the amount of auto-correlation reduces with decreasing size of the least significant part. One of the new encoding techniques exploits this effect.

4.1.1 Game traffic

Q3 is based on a client-server architecture, and messages are asynchronously exchanged over UDP [190]. Hence we analyse client-to-server and server-to-client traffic separately. The client-to-server traffic was taken from trace files collected at CAIA's public game server (see Section 3.1). It contains traffic from 106 clients – some local (same IP subnet) and some remote (non-local). The server-to-client traffic analysis is based on trace files

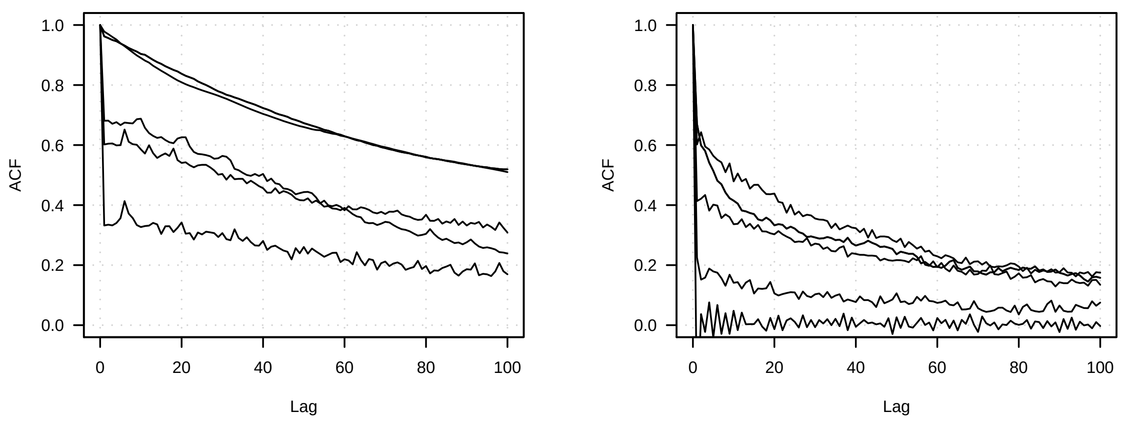


Figure 4.1: Auto-correlation of inter-packet gaps for Q3 client-to-server traffic for local clients (left) and remote clients (right)

captured at a client that connected to 224 different game servers. We limit our analysis to the first 5 000 IPGs of each packet flow.

Figure 4.1 shows the auto-correlation functions (ACFs) for all five local clients and five selected remote clients (histograms are in Appendix C.1). The left graph shows that there is high auto-correlation in a timescale of up to one second, given the average IPG of approximately 10 ms. The two clients with the highest correlation ran Q3 under FreeBSD/Linux, whereas the other three clients ran Q3 under Windows XP. We presume Windows XP introduces more noise and thus the correlation is lower.

Network jitter acts as random noise reducing the auto-correlations introduced at the source. Pearson's correlation of the number of hops between client and measurement point and the sum of the absolute values of the first 100 ACF coefficients is approximately -0.21. However, Figure 4.1(right) shows that even if the traffic is observed many hops away from the source often there is still correlation much higher than the zero correlation of iid IPGs (the top four lines are clients that were 19–21 hops away). For about 10% of the clients the IPGs are uncorrelated, even when the number of hops is relatively small (the bottom line shows a client that was only eight hops away).

Figure 4.2 shows the ACF of server-to-client traffic for a server with very little correlation and the ACF for a server with moderate correlation (histograms are in Appendix C.1). Typically Q3 servers send one packet every 50 ms. However, for about 40% of the servers we observed other patterns, such as a fraction of the IPGs being 100 ms (packet loss or data aggregation by the server) or two peaks around the nominal 50 ms. Often the latter patterns have periodic structures and the auto-correlation is stronger than that shown in Figure 4.2(right). Overall, for approximately 90% of the servers there is at least small auto-correlation of IPGs.

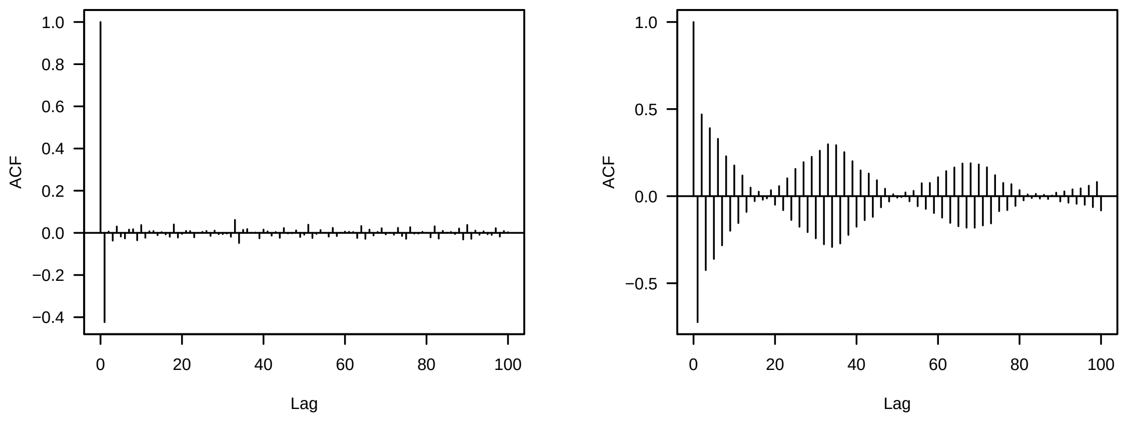


Figure 4.2: Auto-correlation of Q3 server-to-client traffic for a server with little inter-packet gap correlation (left) and a server with moderate correlation (right)

Interestingly, the server with little correlation was located much closer to the measurement point (one hop) than the other server (14 hops). Pearson's correlation of the number of hops traversed and the sum of the absolute values of the first ACF 100 coefficients is close to zero. This suggests that whether there is auto-correlation or not mostly depends on the server and not on the network jitter.

We now analyse how the auto-correlation depends on the size of the least significant part. We computed the ensemble average over the ACFs of all flows for varying least significant parts. The ensemble average at some lag is defined as the mean of the absolute ACF values of each flow at that lag, and referred to as average ACF. As before, we only consider lags up to 100.

Figure 4.3 shows the average ACF of client-to-server and server-to-client traffic both for the full IPGs and decreasing least significant parts. The average auto-correlation of the full IPGs decays more rapidly than individual ACFs, as individual ACFs have lows and highs at different places. Still it is significantly larger than the average auto-correlation for small least significant parts.

4.1.2 Skype

We also analyse the IPGs of Skype based on traffic data collected by Branch *et al.* [192]. The dataset consists of Skype one-to-one calls with varying distance between the peers (6–19 hops). In total the dataset contains 44 traffic flows. As before we only use the first 5 000 IPGs of each flow for the analysis. We do not differentiate between directions as both sides are equal peers.

Figure 4.4(left) shows the ACF of a Skype flow measured at the source. The ACF shows that IPGs are highly correlated and there are periodic structures. The histogram

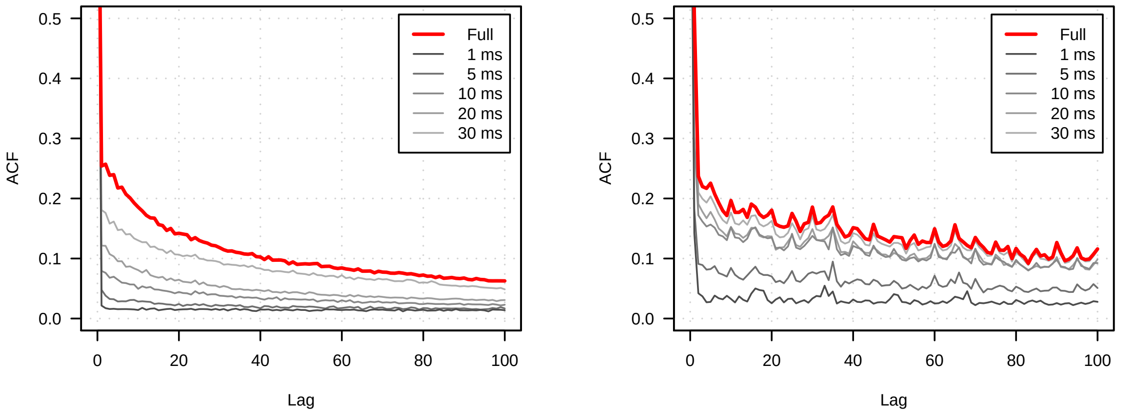


Figure 4.3: Average auto-correlation for Q3 client-to-server traffic (left) and server-to-client traffic (right) (zoomed y-axis)

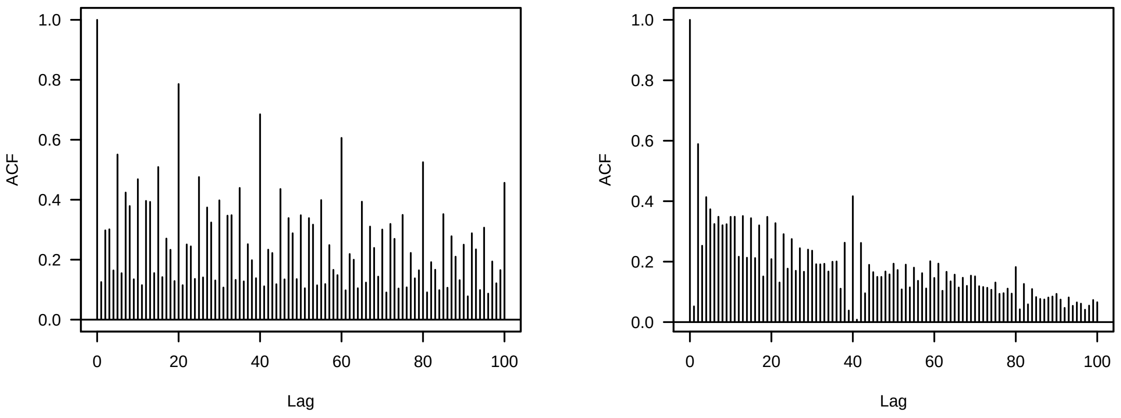


Figure 4.4: Auto-correlation of Skype inter-packet gaps measured at the source (left) and 11 hops away from another peer (right)

has multiple spikes at characteristics times (see Appendix C.1). Figure 4.4(right) shows the ACF of a flow measured for a different peer that was 11 hops away. Despite the higher network jitter, the ACF still shows moderate correlation. However, some traffic from peers further away shows little auto-correlation. Pearson's correlation of the number of hops traversed and the amount of auto-correlation is close to zero.

Figure 4.5 shows the average ACF of the traffic for the full IPGs and different least significant parts. There is high correlation of the full IPGs, but significantly reduced correlation for smaller least significant parts.

4.1.3 UDP and TCP mix

Finally, we analyse a mix of UDP and TCP traffic from the Twente trace (see Section 3.1). Since the trace does not contain payload data we do not know the applications. Based on

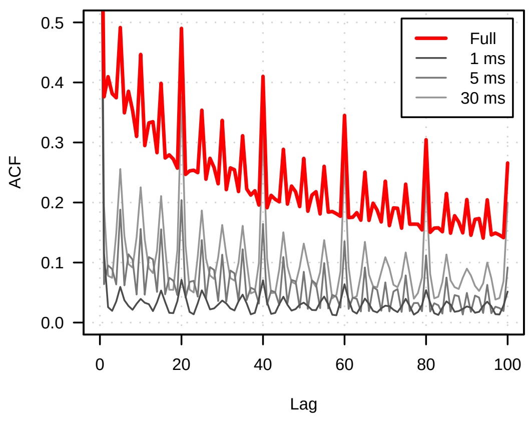


Figure 4.5: Average auto-correlation for Skype depending on the size of the least significant part of the inter-packet gaps (zoomed y-axis)

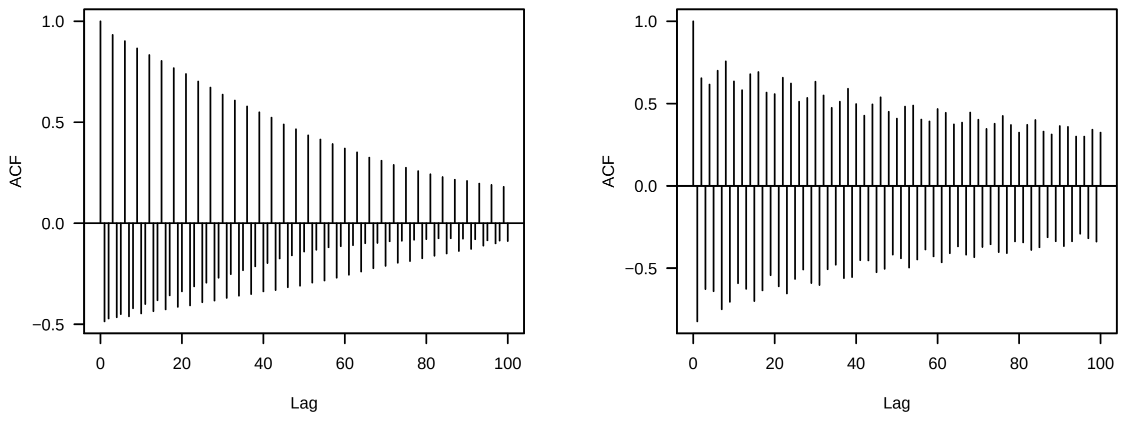


Figure 4.6: Auto-correlation of the inter-packet gaps of data traffic (left) and ACK traffic (right) of a TCP flow (presumably FTP)

an examination of the port numbers it appears that most of the TCP flows were bulk-transfers: HTTP, FTP, NNTP or peer-to-peer file sharing applications, such as Kazaa or BitTorrent. Many UDP flows were game traffic, for example Half-Life/Counterstrike and Quake, but for a significant number of flows we could not identify the applications.

The dataset has 111 UDP and 220 TCP unidirectional traffic flows from different sources. We do not differentiate between different directions of flows because the datasets contain different application types. We analyse the first 5 000 IPGs of longer flows.

Figure 4.6 shows the ACF in the data and ACK direction for a TCP flow observed eight hops away from the originator and three hops away from the other end. This flow has strong auto-correlations. Many other TCP flows have less correlation, but still often it is larger than for iid IPGs. We think for TCP flows the correlations are caused by the cyclical growth and collapse of the congestion window and delayed ACKs. The ACFs of individual UDP flows look similar to the examples shown for Q3.

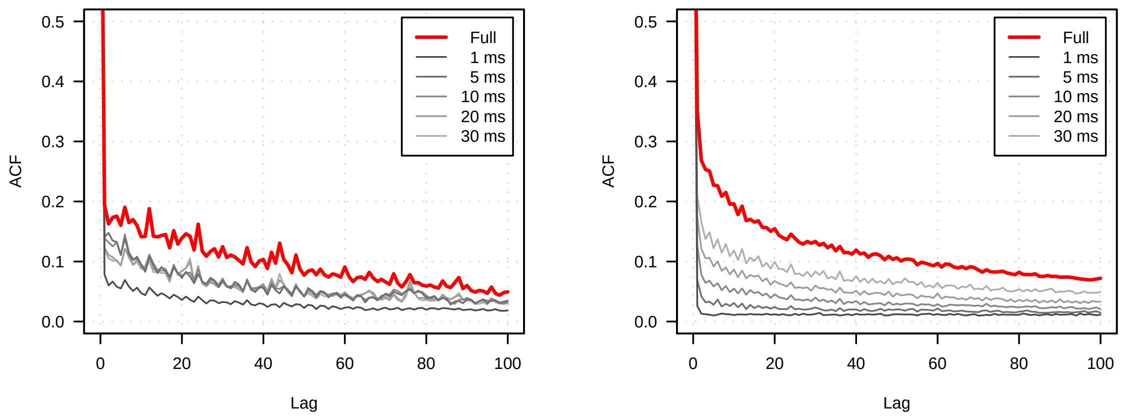


Figure 4.7: Average auto-correlation of inter-packet gaps for TCP traffic (left) and UDP traffic (right) taken from the Twente trace (zoomed *y*-axis)

Figure 4.7 shows the average ACF of the TCP and UDP traffic for the full IPGs and different least significant parts. On average TCP flows have small to moderate auto-correlation and there is still very small correlation even when the least significant part is small. On average UDP flows have stronger auto-correlation than TCP flows and unlike TCP flows there is no auto-correlation for small least significant parts.

Again, there are only very small negative Pearson's correlations between the number of traversed hops and the amount of auto-correlation. The correlation coefficients are approximately -0.05 for the TCP traffic and -0.1 for the UDP traffic.

4.1.4 Conclusions

Auto-correlation of IPGs is caused by the applications and/or the operating systems. Network jitter introduces random noise and reduces the correlations, but even after many hops there often is moderate correlation. Also, if the warden is close to Alice the IPGs are largely unaffected by network jitter.

For the UDP-based applications analysed the IPGs are often moderately to strongly correlated. However, smaller least significant parts of the IPGs are largely uncorrelated. The size of the least significant part where correlation diminishes depends on the application. The TCP flows analysed, which are mostly bulk-transfers, show less correlation, but many flows still have low to moderate correlation.

Note, that our datasets contain no flows where packet transmission times directly depend on users' actions, such as Telnet or interactive SSH, that presumably have iid IPGs. However, in reality these flows are also likely a minority. Furthermore, their packet rate, and hence the maximum transmission rate of covert channels, is much lower than that of other applications, such as bulk-transfer.

4.2 Covert channel

We first review the previous encoding scheme [104, 105] on which our improved schemes are based. We demonstrate that the covert channel is easy to detect when IPGs in normal traffic are correlated. We then propose new improved modulation techniques that keep existing IPG correlations. Finally, we propose a technique for creating passive channels.

4.2.1 Basic encoding scheme

The encoding scheme proposed in [104, 105] enables Alice to produce a covert channel that has exactly the same IPG distribution as normal traffic. Alice and Bob share a model of the IPGs F_{model}, which has been previously generated based on an analysis of real traffic. The model can be a histogram of previously measured IPGs or a standard statistical distribution fitted to the observed IPGs. For example, Gianvecchio *et al.* found that a Weibull distribution fits HTTP IPGs well [104].

Alice and Bob also need synchronised random number generators. They have to agree on a random number generator and a seed value. The seed value can be derived from Alice's and Bob's shared secret. To maximise security a Cryptographically Secure Pseudo Random Number Generator (CSPRNG) should be used [105].

Let $R = r_1, r_2, \ldots, r_n$ be the sequence of Uniform(0,1) random numbers generated and s be a symbol out of the set of possible symbols S. For example, a binary channel has two symbols: $S = \{s_1, s_2\} = \{0, 1\}$. Alice encodes covert bits as follows. Each discrete symbol is transformed into a continuous symbol:

$$F_{\text{cont}}(s, r) = \left(\frac{s}{|S|} + r\right) \bmod 1 = r_{\text{s}} . \tag{4.2}$$

The actual IPGs $d_1, d_2, \ldots, d_n$ are produced by the encoding function:

$$F_{\text{enc}} = F_{\text{model}}^{-1}(r_{\text{s}}) = d , \tag{4.3}$$

where F_{model}^{-1} is the inverse distribution function of the model. Bob decodes the bits from the observed IPGs $\tilde{d}_1, \tilde{d}_2, \ldots, \tilde{d}_n$, which is Alice's generated series modified by timing noise, such as timing inaccuracies at the sender, network jitter and measuring inaccuracies at the receiver. Bob first decodes the continuous symbol by applying:

$$F_{\text{dec}} = F_{\text{model}}\left(\tilde{d}\right) = \tilde{r}_{\text{s}} , \tag{4.4}$$

where F_{model} is the distribution function of the model. Then the discrete symbol received is:

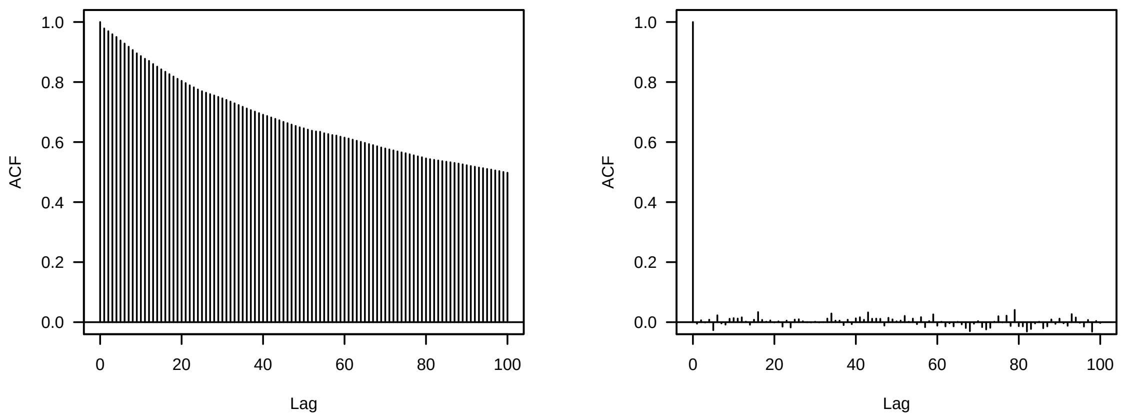


Figure 4.8: Auto-correlation of inter-packet gaps of normal Q3 client-to-server traffic (left) and covert channel with perfect model of the normal traffic (right)

$$F_{\text{disc}}(\tilde{r}_s, r) = |S| \cdot ((\tilde{r}_s - r) \bmod 1) = \tilde{s}\,, \tag{4.5}$$

and in the ideal case the received symbol is identical to the sent symbol ($\tilde{s} = s$).

This encoding scheme generates an IPG distribution that is indistinguishable from the distribution of normal traffic [104, 105]. However, it does not maintain any auto-correlations in the sequence of normal IPGs. This makes the channel detectable because real traffic often has correlated IPGs, as we showed in Section 4.1.

Figure 4.8 illustrates how different the covert channel is from normal auto-correlated traffic. The figure shows the ACF of the IPGs of normal Q3 client-to-server traffic on the left and the ACF of the IPGs of the covert channel on the right. We simulated a binary covert channel using a histogram of IPGs with a bin size of 10 µs as F_{model}.

The IPG histograms of normal traffic and covert channel (not shown) look alike, but the difference of the ACFs is striking. In Section 7.4 we present the results of a more comprehensive analysis of the stealth. We show that the detection accuracy for the covert channel proposed in [104, 105] is very high for all datasets introduced in Section 4.1.

4.2.2 Sparse encoding

We can improve the stealth of the channel by sacrificing capacity. Instead of using all IPGs for encoding, Alice and Bob use only a fraction. Our scheme is also designed so that it is more robust against packet loss than the scheme in [104, 105].

Let f be the fraction of overt packets that carry covert information (called *encoding fraction*). The size of f determines the trade-off between stealth and capacity. Both Alice and Bob generate a sequence of random numbers $T = t_1, \ldots, t_n$ using a CSPRNG and part of the shared secret as seed. All packets where $t_i \leq f$ are selected for the covert channel.

Alice and Bob encode and decode covert bits from the IPGs between selected packets and immediately preceding packets.

This scheme works well if the overt traffic has accessible sequence numbers, but otherwise any packet loss permanently desynchronises Alice and Bob. Furthermore, if Bob is not able to observe the start of the transmission he could never synchronise with Alice. The same problem arises with the encoding scheme in [104, 105]. Any undetected lost packet desynchronises the random numbers between sender and receiver.

The synchronisation problem is solved with the following approach where T and R are computed from the packets themselves. Assume H is a good hash function that maps the inputs as evenly as possible over the output range. Every value in the output range is generated with approximately the same probability (uniform distribution). Let B denote some part of an overt packet that is immutable on the path from Alice to Bob and let b_i be the value of B for the i-th packet. The i-th overt packet is selected if:

$$H(b_i) \leq f\ . \tag{4.6}$$

We assume that the inputs b_i vary sufficiently so that the output of H is approximately uniformly distributed. For active channels Alice can ensure this is the case. For (semi-)passive channels previous work on packet sampling and one-way delay measurement showed that generally this is the case if B is chosen properly, and suggested several suitable choices for H and B [193, 194].

Alice and Bob need to agree on H, B and f. It is possible for Wendy to guess H, B and f and therefore to detect the covert channel, especially as the choices for H and B are limited. To increase security Alice and Bob can use shared key material k_1 and select the packets as follows ($\oplus$ operator denotes the XOR function):

$$H(b_i \oplus k_1) \leq f\ . \tag{4.7}$$

For each packet selected, the sender and receiver compute r_i based on a good hash function H, data from the packet b_i and shared key material k_2:

$$r_i = H(b_i \oplus k_2)\ . \tag{4.8}$$

The outputs of the two hash functions for T and R should be independent, achievable by using different inputs B_T and B_R, or different hash functions H_T and H_R[1].

Algorithm 4.1 shows the encoding algorithm. The inputs of the encode function are the packet itself, the time the previous packet was received, and the time the current packet was received. If the packet is selected for the covert channel the sender computes the IPG

[1]Our proof-of-concept implementation uses two hash functions.

Algorithm 4.1 Sender algorithm for sparse encoding

```
function encode(packet, prev_pkt_time, pkt_time)
  $b$ = hash_input(packet)
  orig_ipg = pkt_time − prev_pkt_time

  if $H_T(b) \leq f$ then
    bits = get_bits()
    $r = H_R(b)$
    delay = $F_{enc}(F_{cont}(\texttt{bits}, r))$
  else
    ipg = orig_ipg

  return ipg
```

Algorithm 4.2 Receiver algorithm for sparse encoding

```
function decode(packet, prev_pkt_time, pkt_time)
  $b$ = hash_input(packet)
  ipg = pkt_time − prev_pkt_time

  if $H_T(b) \leq f$ then
    $r = H_R(b)$
    bits = $F_{disc}(F_{dec}(\texttt{ipg}), r)$
  else
    bits = NA

  return bits
```

based on the covert bits as described in Section 4.2.1. Otherwise, no bits are sent and the IPG is not modified.

The decoding algorithm is simple (see Algorithm 4.2). The receiver checks if the current packet is selected for the covert channel. If yes, the covert bits are decoded as explained in Section 4.2.1. Otherwise, no bits are received.

Figure 4.9 illustrates the effectiveness of sparse encoding based on the same data as Figure 4.8. The covert channel maintains some auto-correlation, but it is reduced compared to the normal traffic. In Section 7.4 we show that sparse encoding is much harder to detect than Section 4.2.1's channel for $f \leq 0.3$. Using only a fraction of IPGs also allows management of the sender's buffering delay for passive channels (see Section 4.2.4).

For active channels sparse encoding requires replacing F_{model} with a more complex model that models a time series' distribution and auto-correlations. For example, autoregressive integrated moving average (ARIMA) models were used to model time series of packet sizes of game traffic more accurately [195]. The sender first uses the model to create a series of IPGs, and then uses the `encode()` function to encode the covert data. For passive channels the correlations are already present in the intercepted traffic.

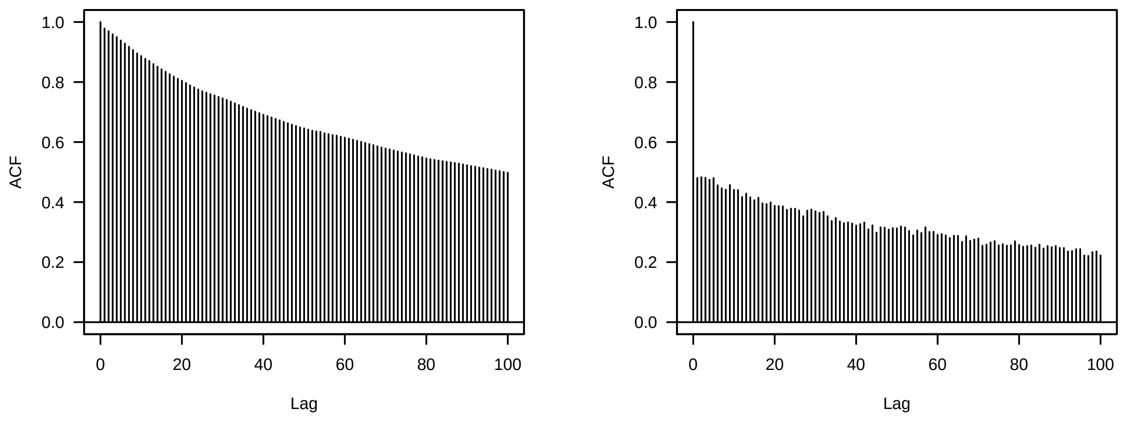


Figure 4.9: Auto-correlation of inter-packet gaps of normal Q3 client-to-server traffic (left) and covert channel using sparse encoding with $f = 0.3$ (right)

4.2.3 Sub-band encoding

Sparse encoding improves the stealth of the channel, but the capacity is severely reduced. Now we present a different modulation scheme that provides a higher capacity and is also better in maintaining auto-correlation of IPGs. However, it is less robust. The scheme encodes covert bits into the least significant part of IPGs as defined in Section 4.1.

Let l be the size of the least significant part of the IPGs in micro- or milliseconds. The parameter l determines the trade-off between stealth and robustness. Let D be the range of IPGs (maximum minus the minimum). Then an IPG distribution typically spans many *sub-bands* of size l, precisely $m = \lceil D/l \rceil$ bands, although in the extreme case it may fit in only a single sub-band. The location of the sub-bands, the IPG value marking the start of each band, depends on the start value of the first band. The location must be selected carefully to minimise the error rate for a given IPG distribution.

For each of the sub-bands the basic encoding scheme described in Section 4.2.1 is used. However, now we have one probability distribution of the least significant part of the IPGs for each sub-band j, resulting in $F^{(j)}_{\text{enc}}(.) = F^{-1(j)}_{\text{model}}(.)$ and $F^{(j)}_{\text{dec}}(.) = F^{(j)}_{\text{model}}(.)$. Since the scheme encodes bits using sub-bands we named it sub-band encoding.

When building the model from the example traffic a distribution of the least significant part of the IPGs must be estimated for each sub-band. If the example traffic is only a small sample and the number of bands is large it may happen that there are only a few or even zero values for some bands. Our algorithm augments the data for these sub-bands with uniformly distributed random values so that a minimum number of samples is reached[2].

Algorithm 4.3 shows the encoding algorithm. Let BL be a set of tuples $\left(d^{(j)}_{\text{start}}, x_j\right)$ that associate each sub-band index x_j with an absolute IPG value $d^{(j)}_{\text{start}}$ marking the start of the

[2]For sub-bands outside the range covered by the example traffic we also assume uniform distributions.

band (base IPG). We assume that BL has a uniform band model for all bands not covered by the example traffic. First, the algorithm determines the actual sub-band based on the original unmodified IPG. Then the modified IPG is the sum of the least significant part as given by the sub-band model and the base IPG of the sub-band.

Algorithm 4.3 Sender algorithm for sub-band encoding

```
function encode(packet, prev_pkt_time, pkt_time)
  $b$ = hash_input(packet)
  orig_ipg = pkt_time − prev_pkt_time

  base_ipg = $\lfloor$orig_ipg$/l\rfloor \cdot l$
  band = BL[$d_{\text{start}}$ = base_ipg].$x$
  bits = get_bits()
  $r = H_{\text{R}}(b)$
  ipg = $F_{\text{enc}}^{(\text{band})}(F_{\text{cont}}(\text{bits}, r))$ + base_ipg

  return ipg
```

Algorithm 4.4 shows the decoding algorithm. The receiver selects the sub-band based on the observed IPG. Then it computes the least significant part of the IPG and decodes the bits as before.

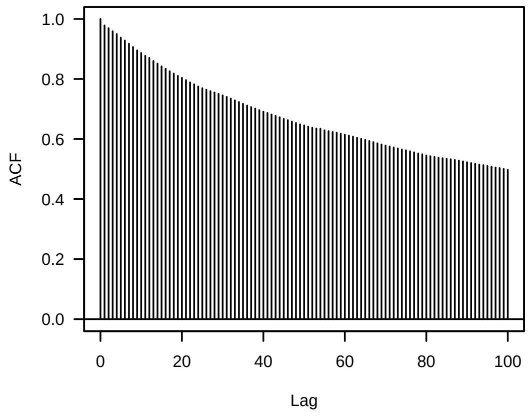
Algorithm 4.4 Receiver algorithm for sub-band encoding

```
function decode(packet, prev_pkt_time, pkt_time)
  $b$ = hash_input(packet)
  ipg = pkt_time − prev_pkt_time

  base_ipg = $\lfloor$ipg$/l\rfloor \cdot l$
  lsp = ipg − base_ipg
  band = BL[$d_{\text{start}}$ = base_ipg].$x$
  $r = H_{\text{R}}(b)$
  bits = $F_{\text{disc}}\left(F_{\text{dec}}^{(\text{band})}(\text{lsp}), r\right)$

  return bits
```

Figure 4.10 illustrates the effectiveness of sub-band encoding based on the same data as Figure 4.8. The covert channel only very slightly decreases the auto-correlations. Our results in Section 7.4 show that sub-band encoding is harder to detect than sparse encoding while providing a significantly higher capacity.

The drawback of sub-band encoding is that it is less robust against network jitter, but network jitter is often relatively small even on paths with many hops if there is no congestion. However, the scheme is less robust against an active warden that introduces artificial jitter by re-timing the overt packets.

For active channels sub-band encoding requires replacing F_{model} with a model that also captures the auto-correlations, as discussed in Section 4.2.2. The sender uses the



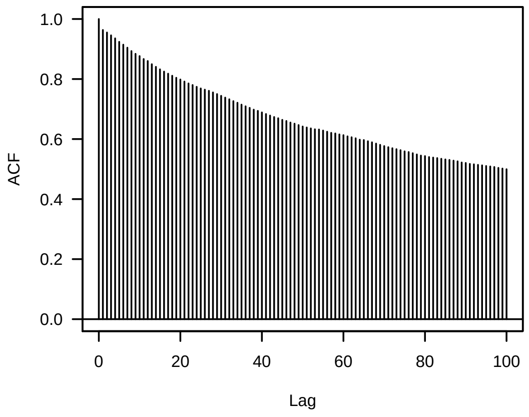


Figure 4.10: Auto-correlation of inter-packet gaps of normal Q3 client-to-server traffic (left) and covert channel using sub-band encoding with l = 5 ms (right)

model to create a series of IPGs, and then uses the `encode()` function to encode the data. For passive channels the correlations are already present in the intercepted traffic.

4.2.4 Passive channel

The covert channel in [104, 105] was developed as active channel, where the covert sender generates ‘fake’ application traffic with the appropriate timing. Creating fake UDP traffic is simple, especially if both directions of a flow are ‘independent’. However, creating fake TCP connections is more complicated, and presumably that is the reason why the previous proof-of-concept implementation used UDP [196].

Active channels are simpler to implement than passive channels, but they require the covert sender to imitate normal traffic very well. The sender must emulate the application protocols to create realistic looking messages, or the covert channel could be easily detected by packet inspection. It is possible to imitate real protocols, but an implementation is cumbersome and the support of a larger number of protocols is difficult.

Also, the covert sender must ensure that traffic patterns, such as message sizes and timing, look realistic. For example, if the covert channel hides in HTTP, as suggested in [104], simple patterns such as repeatedly transferring the same object(s) would be suspicious. Instead a realistic looking pattern of HTTP requests must be generated. Previous work [104, 105] did not address how to introduce variance into the model to prevent all covert channels from looking identically.

To avoid these issues we propose to use passive or semi-passive channels, where the traffic of real applications is used as cover. The covert sender either uses existing traffic of unwitting users or generates the overt traffic using real applications. This guarantees that real application protocols are used and the traffic patterns are realistic.

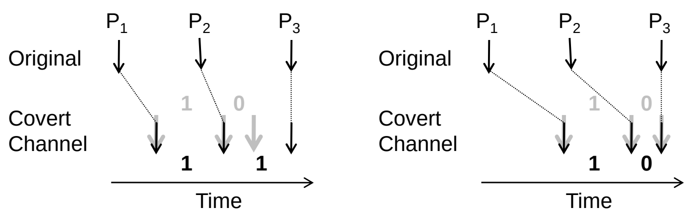


Figure 4.11: Insufficient buffering causes a wrong bit to be sent (left), but with more initial buffer delay both bits are sent correctly (right)

The disadvantage of passive channels is the delay added to the overt traffic due to the buffering necessary for changing the timing of packets, as illustrated in Figure 4.11. The figure shows three example packets, whose timing is changed by the covert sender to encode the bits "10" (logical zero/one encoded as small/large IPG). In Figure 4.11(left) the initial buffering delay is insufficient to encode the zero and instead a one is sent. Figure 4.11(right) shows that with more initial delay the bits are sent correctly.

Algorithm 4.5 shows the passive sending technique. The first packet is delayed for a pre-defined time (`INITIAL_DELAY`). For each following packet the algorithm checks if the IPG between the current and previous packet is used for encoding bits or for buffer management purposes. This packet selection is identical to the selection used by sparse encoding, except that the threshold is $\hat{f}$ is usually much larger than f [3].

If bits are encoded the sparse encoding or sub-band encoding `encode()` function is called. If the computed IPG is smaller than the time already elapsed between the reception of the current packet and the sending of the previous packet, it is not possible to produce the desired IPG due to insufficient buffering. Otherwise the computed IPG is adjusted according to the time the last packet was or will be sent.

If an IPG is not used to encode bits, it is not modified if the current buffering delay is within a target range (`MIN_DELAY`, `MAX_DELAY`). If the current buffering delay is outside the target range the original IPG is adjusted to get into the target range. The algorithm aims to keep the buffering delay under a specified maximum to reduce latency and over a specified minimum to avoid bit errors caused by insufficient buffering.

We assume that extreme IPG values are removed from F_{model}, i.e. the lower and upper x-percentiles are removed. Then a simple way of adjusting the delays is to randomly choose IPGs from the small and large end of F_{model}. The frequency of adjustments is at most $1 - \hat{f}$ and with a larger target delay range it is usually significantly smaller. This means the impact on the shape of the IPG distribution is usually very small; however it may be detectable, for example revealed by the first-order entropy.

A stealthier but more complicated technique works as follows. Let U be a list of n IPG values randomly selected from F_{model}. Each time an adjustment is required, the

[3]In our experiments we set $\hat{f} = 0.95$.

Algorithm 4.5 Passive covert sender algorithm

```
function send(packet, prev_pkt_time, prev_pkt_send_time, pkt_time)
  if prev_pkt_time ≠ NA then
    $b$ = hash_input(packet)
    orig_ipg = pkt_time − prev_pkt_time
    offset = pkt_time − prev_pkt_send_time

    if $H_T(b) \leq \hat{f}$ then
      ipg = encode(packet, prev_pkt_time, pkt_time)
      if ipg ≥ offset then
        ipg = ipg − offset
      else
        // insufficient buffering!
        ipg = 0
    else
      ipg = orig_ipg
      if offset < −MAX_DELAY then
        // decrease buffering delay
      if offset > −MIN_DELAY then
        // increase buffering delay
  else
    ipg = INITIAL_DELAY

  prev_pkt_time = pkt_time
  prev_pkt_send_time = pkt_time + ipg

  return ipg
```

value in U closest to the required IPG, based on the current buffering delay and the target range, is selected. This value is then removed from U. Once U is empty it is refilled with another n values randomly chosen from F_{model}. Since U is a random sample of F_{model}, the adjustments do not change the shape of the IPG distribution. However, the buffer management becomes more difficult. We leave this approach as future work.

The buffering delay introduced potentially reduces the performance of the applications generating the overt traffic. However, for UDP-based flows or low-rate TCP flows a performance reduction may not be noticeable. In Section 4.5 we analyse the delay and show that it can be reduced by allowing a higher bit error rate. Furthermore, for semi-passive channels there is no issue as the covert sender is also the ‘user’ of the applications and hence unconcerned about additional delay.

4.3 Implementation considerations

First we discuss issues arising when using the improved passive timing channel with certain applications and propose solutions. Next, we discuss measures to reduce timing noise introduced at the sender and receiver. Finally, we describe the choice of hash functions and their input, and verify that the output is indeed uniformly distributed.

4.3.1 Overt traffic dependencies

A number of issues arise with the improved passive timing channel and Section 4.2.1's active timing channel that were not identified in [104, 105], because the proof-of-concept implementation [196] or simulation [105] was based on IP packets and did not consider the effects of transport protocols or applications.

Section 4.2.1's encoding scheme [104, 105] randomises the IPGs completely. This is problematic if an IPG distribution contains very large values, which can be the case for human-driven traffic, such as interactive SSH. For example, if a covert sender assigns a very long IPG to an 'unsuitable' packet pair, such as the SYN and ACK of a TCP session's initial handshake, this would be suspicious for an active channel and even worse could entirely disrupt the overt traffic for a passive channel (TCP session setup timeouts).

For passive channels very long IPGs cause bit errors unless the buffering delay is very large. But very large buffering delays are impractical, and hence if very long IPGs are common, such as for interactive SSH, the overall bit error rate will significantly increase. Therefore, our passive channel does not use very large IPGs for encoding.

IPGs over a threshold are removed from F_{model}. Covert sender and receiver ignore all IPGs over the threshold, i.e. they do not encode or decode. To avoid further synchronisation problems the sender and receiver make the decision based on IPGs computed from TCP timestamps, which are widely used these days and immutable in the network.

Covert sender and receiver must process the packets in the correct order, based on the TCP timestamps, before computing the IPGs. We assume that any reordering between actual sender and covert sender is very small, because typically they are very close. The receiver buffers and reorders copies of packets. Hence this mechanism does not introduce additional delay.

Our approach also avoids TCP handshakes failing due to timeout. On the other hand it lowers the capacity compared to active channels where large IPGs encode bits. Our technique works for UDP-based protocols that have per-packet timestamps, such as RTP.

Bidirectional flows associated with request/response protocols can also pose a challenge for active and passive timing channels. For example, a TCP client may have to wait for a response or a TCP ACK before it can send the next packet. These dependencies lead to errors if two timing channels are encoded in both directions of a flow. Furthermore, they are problematic for passive channels. If the overt sender is waiting for a response before continuing to send, the covert sender's buffer is drained completely. However, for some UDP-based applications, such as Q3, the packet flows in both directions are basically independent (asynchronous).

For bidirectional asynchronous protocols the passive channel is encoded in both directions. However, for TCP only one direction of a flow is used. If a bidirectional covert channel is needed, two simultaneous TCP flows could be used. We leave the study of bidirectional channels over single TCP flows as future work.

If the overt traffic contains timestamps that are accessible for the warden, such as TCP timestamps, (semi-)passive covert senders must manipulate these timestamps according to the modified IPGs. Otherwise the channel could be detected by observing discrepancies between the protocol timestamps and the actual IPGs. If the covert sender cannot manipulate these timestamps it must keep IPG modifications small so they look like noise.

4.3.2 Sender/receiver timing accuracy

Channel timing noise consists of packet timing inaccuracies at the covert sender, packet timestamping inaccuracies at the covert receiver and network jitter. There are several things covert sender and receiver can do to minimise inaccuracies.

The receiver noise can be minimised by using high-performance capture cards that provide highly accurate packet timestamps, such as DAG cards [197]. Although high-performance capture devices exist, most covert receivers will rely on consumer-grade network interface cards (NICs) that may or may not support timestamping in hardware.

Our prototype based on CCHEF, described in Appendix A, was carefully designed to maximise the sender's packet timing accuracy (see Appendix C.2). However, CCHEF is a userspace program and thus competes with other userspace programs for CPU time. Other programs using a lot of CPU time, such as the Q3 client, decrease CCHEF's timing accuracy. To avoid this we use real-time Linux [198] and CCHEF runs as real-time process with high priority (see Appendix C.2). Furthermore, we set the kernel's tick frequency to 10 kHz to minimise the size of time slices[4].

We performed experiments to estimate the sender's timing accuracy (see Appendix C.2). The results show that for Q3 client-to-server traffic, with a packet rate of approximately 91 pps, the error is less than 40 µs, despite the covert sender and Q3 client machine running at 100% CPU load. For scp traffic the error is higher because of the higher packet rate (approximately 164 pps) and the larger amount of processing needed for TCP packets, but still mainly within 100 µs.

Some of the above measures to improve the covert sender's timing accuracy imply a high degree of control over the host. However, if CPU utilisation and packet rate are not very high they are not required. For example, using interactive SSH or Q3 server-to-client traffic in our testbed does not require a real-time operating system with high tick frequencies to achieve low error rates.

[4]Even high frequencies cause only relatively small context switching overhead on modern CPUs [199].

4.3.3 Hash functions

We use the CRC32 and “Bob’s hash” hash functions[5]. For UDP we use 40 bytes of each packet as input (IP ID, source and destination address, protocol, total length, UDP header, first 19 bytes of UDP payload) as suggested in [193, 194]. However, for TCP only 21 bytes are used, because the covert sender modifies the TCP timestamp as well as the TCP checksum. This means the IP header fields listed above, and the TCP sequence number and acknowledgement number are used.

We verified that the output of the hash functions is uniform given the input in our experiments described in Section 4.5. CDFs of the values of the two hash functions show that for all applications both functions deliver almost perfect uniform distributions (see Appendix C.3). We also used the two-sample Kolmogorov-Smirnov (KS) test to test the hypothesis that the different distributions are uniform. In each test we compared the generated hash values with a uniform distribution. Based on the results in all cases we cannot reject the hypothesis that the distributions are uniform at 95% significance level.

4.4 Channel capacity

Now we develop a model for the channel and derive the channel capacity. We use the results in Section 4.5 to estimate the maximum transmission rates of the channel depending on different networks conditions.

The channel model of the IPG timing channel is similar to the model of the TTL channel described in Section 3.3. Again, we assume the channel is memoryless and we focus on a binary channel. There are three possible sources of errors:

- bit substitutions caused by timing jitter,
- bit substitutions and deletions caused by loss of overt packets and
- bit substitutions caused by reordering of overt packets.

Timing jitter, the combined effects of packet timing inaccuracies, timestamping inaccuracies and network jitter, causes bit substitutions on the channel. We can model the channel under the effects of timing jitter either as a BSC [22] or a BAC [169]. In our experiments the resulting error rates are approximately symmetric. Hence we use the BSC to determine the capacity. If covert sender and receiver can utilise information contained in the overt traffic, the impact of the last two sources of errors depends on the protocol.

If the protocol has no sequence numbers (e.g. UDP), the covert receiver does not know which bits were lost. Hence we model packet loss as a binary deletion channel [22]. If the

[5]Previous research showed that CRC32 and “Bob’s hash” are suitable for packet selection [194].

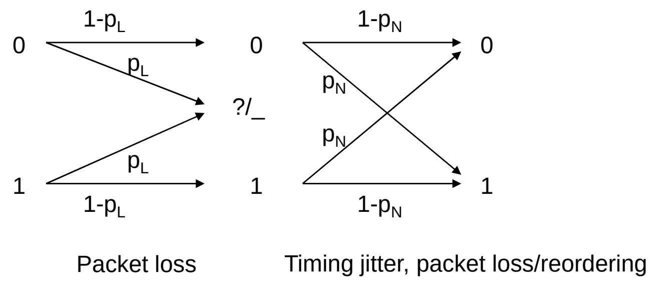


Figure 4.12: Inter-packet gap timing channel model

transport protocol has sequence numbers (e.g. RTP or TCP), the covert receiver knows which packets were lost and we use a binary erasure channel [22]. In contrast to storage channels, like the TTL channel, packet retransmissions (e.g. TCP) are only helpful if they can be manipulated such that their IPGs encode the lost bits.

Since the covert channel is encoded in IPGs between two packets, every lost packet not only causes a deletion or erasure but also a possible substitution error in the following bit, effectively further increasing the timing jitter. Packet reordering always results in potential substitution errors because even if the covert receiver can reconstruct the original sequence of packets it does not know the original IPGs. Effectively packet reordering also increases the timing jitter. The error rate of the BSC is $p_\mathrm{N} = f(p_\mathrm{T}, p_\mathrm{R}, p_\mathrm{L})$, where p_T is the substitution error rate caused by timing jitter, p_R is the substitution error rate caused by reordering, and p_L is the packet loss rate. We empirically measure p_N in Section 4.5.

We model the overall channel as a cascade of the two separate channels where the leftmost channel is either a deletion channel with symbol lost indicated by a "_" or an erasure channel with a symbol value unknown indicated by a "?" (see Figure 4.12).

The capacities for the individual channels and the cascade were derived in Section 3.3. In any case the lower bound of the capacity of the IPG covert channel is:

$$C \geq \max\{0, 1 - [H(p_\mathrm{L}) + (1 - p_\mathrm{L})H(p_\mathrm{N})]\} \ . \tag{4.9}$$

If packet loss can be detected based on information in the overt traffic the capacity is:

$$C = (1 - p_\mathrm{L})(1 - H(p_\mathrm{N})) \ . \tag{4.10}$$

If the IPGs of retransmitted packets can be used to retransmit lost bits the capacity is:

$$C = 1 - H(p_\mathrm{N}) \ . \tag{4.11}$$

Table 4.1 summarises the channel capacity for the UDP and TCP transport protocols. In contrast to the TTL channel there is no gain for encoding in single flows. The capacity C is always in bits per IPG (bits per symbol) and the average transmission rate in bits per second is computed using Equation 3.14, where f_S is the average rate of packet pairs.

Table 4.1: Channel capacity based on overt traffic

	UDP w/o seq numbers	**UDP with seq numbers**	**TCP**
Single or multiple flows	Equation 4.9	Equation 4.10	Equation 4.11

4.5 Empirical evaluation

Here we describe our testbed, present the measured channel error rates and resulting capacities based on various network conditions, and evaluate the throughput achieved using Section 3.4's protocol for reliable transport relative to the channel capacity. Finally, we analyse the delay introduced into the overt traffic for passive channels.

4.5.1 Methodology

Our testbed consisted of two computers connected via a Fast Ethernet switch[6]. Alice and Bob, CCHEF instances, were co-located with the actual sender and receiver. However, they could have been middlemen acting without knowledge of the user(s) of the applications. As applications we used scp, interactive SSH and Q3, the same applications that were used in the TTL channel experiments (see Section 3.5.6).

Each experiment lasting 20 minutes was repeated three times and we report the average statistics. Network delay/jitter, packet loss and reordering were emulated using Linux Netem [189]. The network delay/jitter was emulated using Pareto distributions with a mean of 25 ms and standard deviations (σ) of 0, 0.1, 0.2, 0.3, 0.5, 1 and 2 ms in each direction, since previous research suggested that network jitter is heavy-tailed [200], and Netem only supports Uniform, Gaussian and Pareto jitter distributions. Setting the kernel's tick timer frequency to 10 kHz ensured delay emulation was accurate to ±100 µs.

Figure 4.13 shows CDFs of the absolute IP Delay Variation (IPDV [201]), both in the testbed with Pareto distributions with different standard deviations and measured across two Internet paths. The 8-hop Internet path's RTT was approximately 32 ms, and the 13-hop path's RTT was approximately 46 ms. All measurements used ping (ICMP request/reply), potentially inflating the observed Internet path RTTs, as pinged hosts were not under our control and ICMP echo packets may have low priority in routers. We estimated the one-way delay to half the measured RTT.

We emulated packet loss and reordering rates of 0%, 0.1%, 0.5% and 1% with a correlation of 25% [189], because loss and reordering in the Internet are typically bursty. Prior to the experiments we verified the accuracy of Netem (see Appendix F). Due to the

[6]An Intel Celeron 2.4 GHz with 256MB RAM and an Intel Celeron 3.0 GHz with 1GB RAM, both running LinuxRT 2.6.20. A low-end Alloy NS-16J switch introduced timing errors up to 100–200 µs.

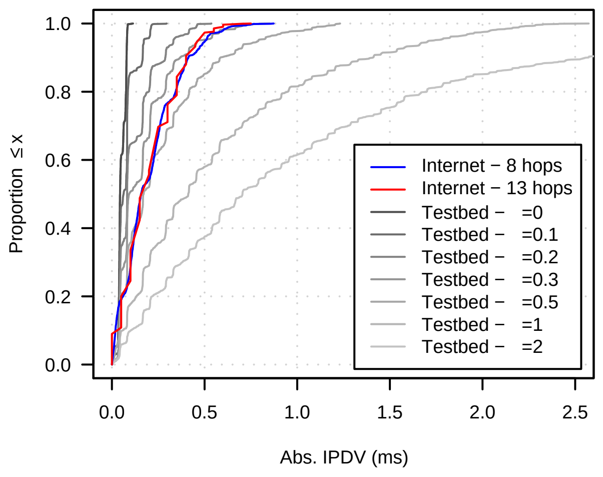


Figure 4.13: Absolute IPDV distributions for different testbed settings and two real paths across the Internet

large number of parameters we do not explore all parameter combinations but investigate the dependency on one parameter while the others are fixed to sensible values.

The models F_{model} were built as follows. First, we measured the IPG distribution of each application at the source, unaffected by timing jitter. We then added a small amount of noise. Without the added noise the covert channel would not work well for applications with very narrow IPG distribution, such as Q3 client-to-server and scp traffic. The added noise represents timing jitter caused by the network, or a high CPU or NIC load at the host, which the warden would also encounter in reality, especially when being multiple hops away from the covert sender. For applications with wider IPG distributions, such as Q3 server-to-client and SSH traffic, it may not be necessary to add noise. However, to even the playing field we always added noise.

For Q3 and SSH we used normally distributed noise. For scp there are many very small IPGs between data packets (tens of microseconds up to a few hundred microseconds) and fewer larger IPGs (tens of milliseconds). Since accurate packet timing in the order of a few tens of microseconds is impossible, the model was constructed such that the IPGs between 'back-to-back' packets were increased (using a uniform noise distribution) and the larger IPGs were reduced to maintain the original rate.

For the noise we used standard deviations of 0.5 ms, 0.75 ms and 1 ms (referring to a model with noise standard deviation of x as model-x). For Q3 and SSH traffic the location of the sub-bands was chosen such that peaks in the distributions are approximately in the middle of bands. Our models are histograms with small bin sizes of 10 µs, as our traffic sources cannot be modelled well with standard statistical distributions.

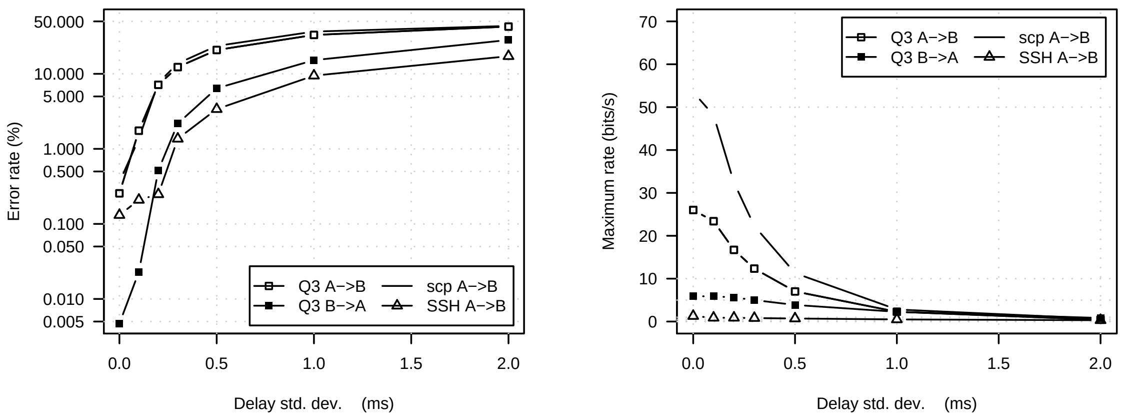


Figure 4.14: Error rate (left) and maximum transmission rate (right) for sparse encoding with model-0.5 (left graph has log *y*-axis)

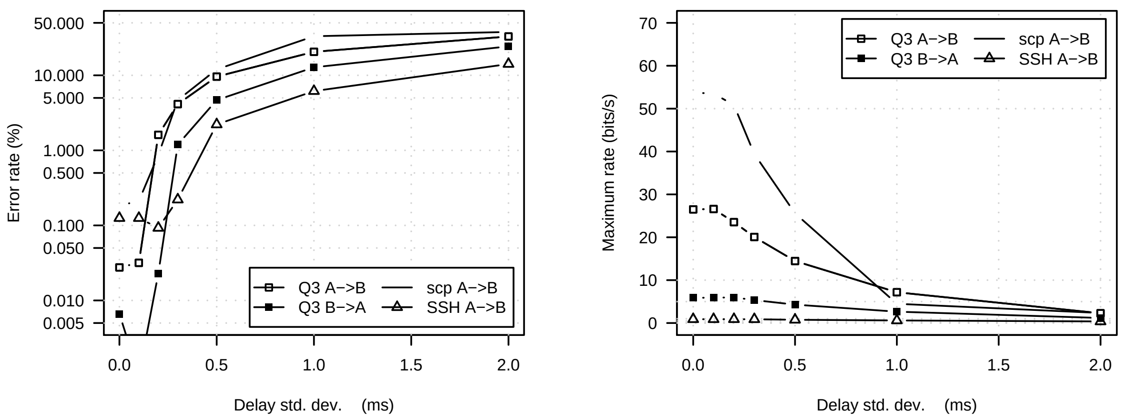


Figure 4.15: Error rate (left) and maximum transmission rate (right) for sparse encoding with model-1.0 (left graph has log *y*-axis)

4.5.2 Error rate and capacity

First we measured the error rate of the channel based on the encoding scheme, network jitter, packet loss and reordering. We also investigated the effect of different models (model-0.5, model-0.75, model-1.0). Based on the channel model we computed the maximum transmission rates. For sparse encoding we used $f = 0.3$ because the channel is hard to detect for such low f (see Section 7.4). For sub-band encoding we set $l = 5$ ms. The buffering parameters were set to minimise errors caused by insufficient buffering.

Figure 4.14 and Figure 4.15 show the error rate for sparse encoding for model-0.5 and model-1.0 depending on the standard deviation of the emulated delay without packet loss or reordering (graphs for model-0.75 are in Appendix C.4). They also show the maximum transmission rates. There is not much variation in the measured error rate, except for SSH with error rates below 1%, and hence for clarity no error bars are shown.

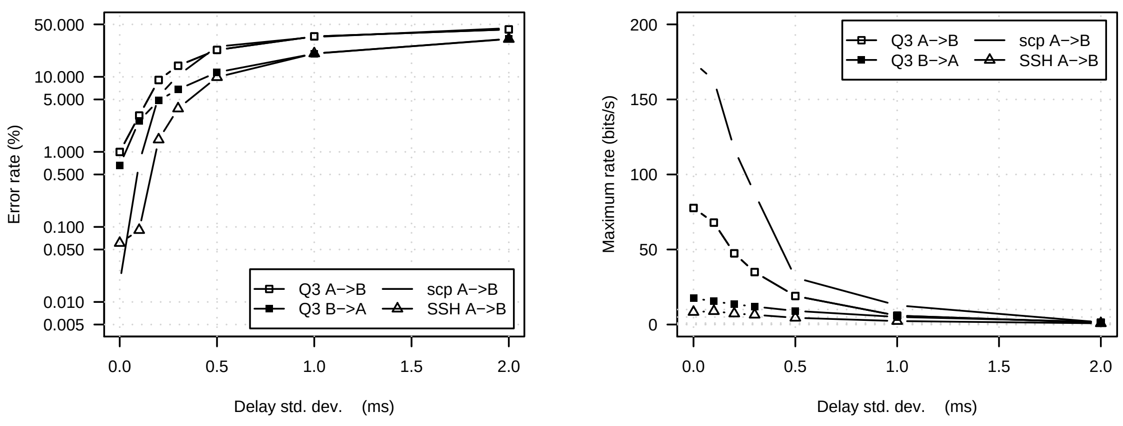


Figure 4.16: Error rate (left) and maximum transmission rate (right) for sub-band encoding with model-0.5 (left graph has log y-axis)

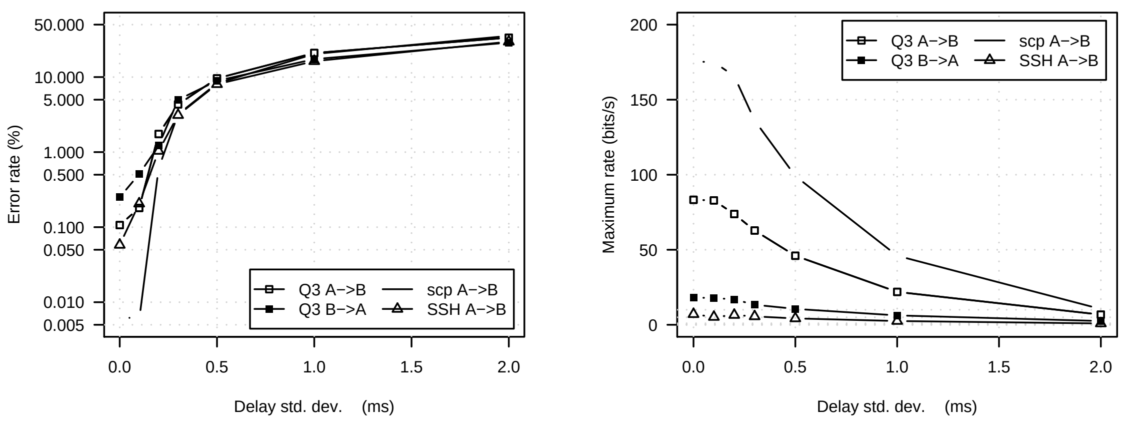


Figure 4.17: Error rate (left) and maximum transmission rate (right) for sub-band encoding with model-1.0 (left graph has log y-axis)

The results show that for model-0.5, the error rate increases relatively quickly with increasing network jitter. But for model-1.0 the increase is more moderate. The error rate for Q3 server-to-client traffic and SSH is much smaller due to the wider model distributions, but the maximum transmission rates are still small because of the low packet rates. Due to $f = 0.3$ the maximum rate of sparse encoding is already much lower than for sub-band encoding or the TTL channel.

Figure 4.16 and 4.17 show the same graphs for sub-band encoding. We see that sub-band encoding is less robust than sparse encoding; the error rate increases faster with increasing network jitter. Again, Q3 server-to-client traffic and SSH have the lowest error rates, but also the smallest transmission rates. Because of the larger number of overt packets used the maximum transmission rates are significantly larger for small to moderate jitter compared to sparse encoding.

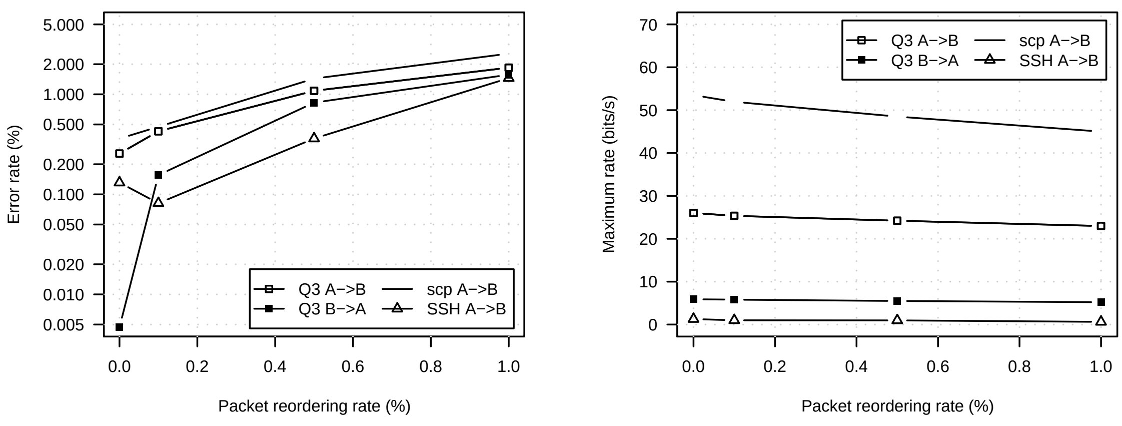


Figure 4.18: Error rate (left) and maximum transmission rate (right) depending on packet reordering for sparse encoding with model-0.5 (left graph has log y-axis)

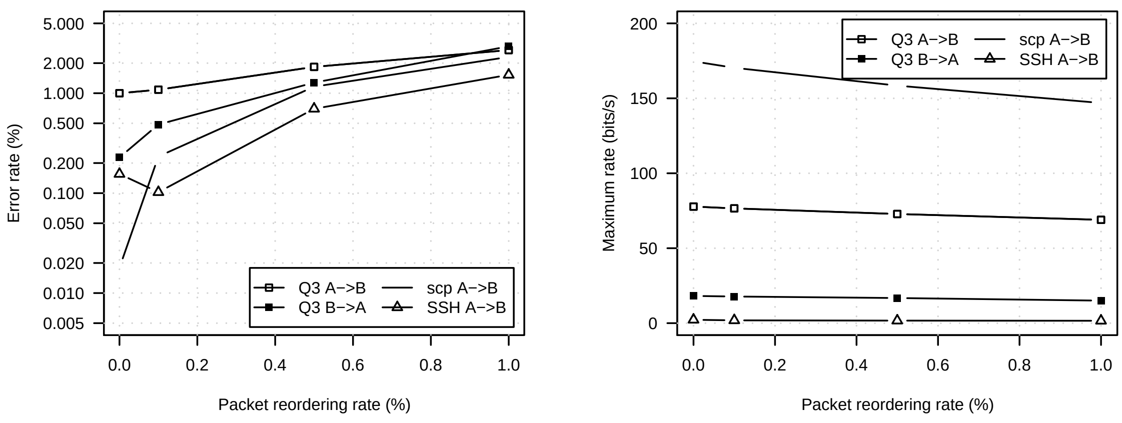


Figure 4.19: Error rate (left) and maximum transmission rate (right) depending on packet reordering for sub-band encoding with model-0.5 (left graph has log y-axis)

We now investigate the effects of packet reordering. The packet reordering rate was configured via Netem, but the actual error rate also depends on the delay configured and the application's IPG distribution (see Section 3.5.6). We configured the delay individually for the different applications so that the resulting reordering rates were similar.

Figure 4.18 shows the error rates and maximum transmission rates for sparse encoding depending on the packet reordering rate for model-0.5 without emulated network jitter. Figure 4.19 shows the same for sub-band encoding.

For sub-band encoding the bit error rate approximately equals the expected reordering rate from Equation 3.4 (differential case) plus the errors caused by timing jitter. However, for sparse encoding the error rate is lower because in Equation 3.4 we assumed that all packet pairs are used.

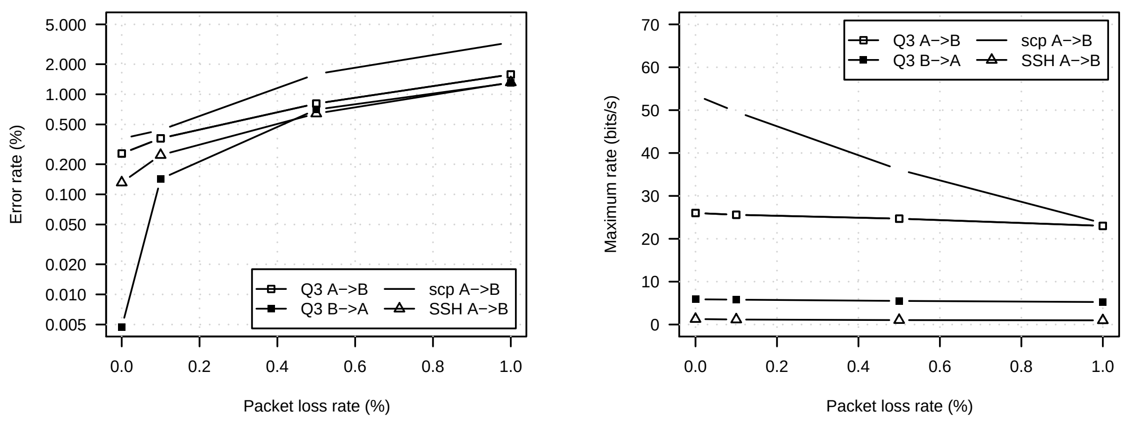


Figure 4.20: Error rate (left) and maximum transmission rate (right) depending on packet loss for sparse encoding with model-0.5 (left graph has log y-axis)

The maximum transmission rates are only slightly reduced for the reordering rates emulated. We also performed experiments with emulated network jitter ($\sigma = 0.2$ ms) and packet reordering. Normalised on the base error rate with zero packet reordering the error rates are similar to the rates without emulated jitter. The error rates produced for different models are also similar.

Finally, we investigate the effects of packet loss. Figure 4.20 shows the error rates and maximum transmission rates for sparse encoding depending on the packet loss rate for model-0.5 without emulated network jitter. Figure 4.21 shows the same for sub-band encoding.

We computed the error rates as follows. After each experiment the lost packets were identified using the packet hashes. Then dummy zero bits were inserted for deletions. Since the data was uniform random approximately 50% of the inserted bits cause errors. The remaining errors, minus the base error at zero packet loss, are the substitution errors following each deletion. Unlike for the TTL channel with AMI encoding, it is difficult to predict these errors because they depend on the application's IPGs.

For Q3 and SSH the transmission rate decreases only slightly with increasing packet loss. For scp the maximum transmission rate decreases more substantially, because the packet rate of the overt traffic decreases. TCP flows usually experience reduced packet rates in the presence of network delay and packet loss, but for passive IPG timing channels this effect is increased because of the added buffering delay. As for reordering, the resulting error rates are smaller for sparse encoding compared to sub-band encoding. Again, the error rates caused by packet loss do not change significantly with different emulated network jitter or different models.

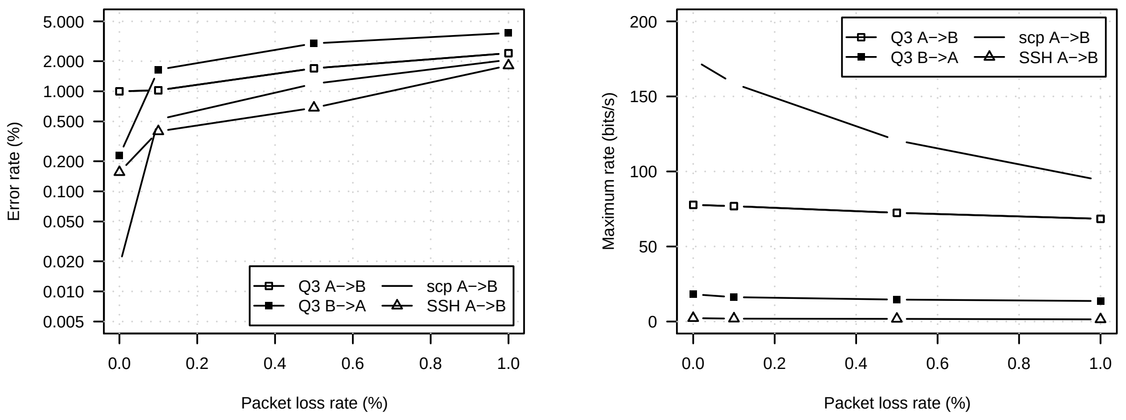


Figure 4.21: Error rate (left) and maximum transmission rate (right) depending on packet loss for sub-band encoding with model-0.5 (left graph has log y-axis)

Our new timing channels have high capacities if network jitter is low. Transmission rates are up to over hundred bits per second. However, if the network jitter is high the capacity is severely reduced and much smaller than the TTL channel's capacity.

Sparse encoding has lower error rates since each bit is encoded using the full distribution. For larger f the error rate is similar, but the maximum transmission rate increases accordingly. Sub-band encoding provides higher transmission rates because a larger fraction of the IPGs can be used for encoding. The larger the sub-band size the more robust the encoding becomes. Sub-band encoding can be used if the reduction in robustness is tolerable, i.e. when there is not too much timing jitter. Otherwise, sparse encoding should be used since it is more robust.

Q3 and scp have very narrow IPG distributions, and in our experiments even interactive SSH was limited because we played back a short recorded session repeatedly. Applications with wider IPG distributions can provide much more robust channels. However, transmission rates would still be low since such applications also have low packet rates. Active channels have slightly higher transmission rates because no IPGs need to be 'sacrificed' for buffer management.

4.5.3 Burstiness of errors

The burstiness of errors does not affect the channel capacity, but it affects the performance of techniques for reliable data transport. How bursty the errors are depends on the encoding scheme and the IPG distribution of the normal traffic. Figure 4.22 illustrates this for scp and Q3 showing CDFs of the distance between errors in bits for the experiments with $\sigma = 0.3$ ms and no packet loss and reordering and simulated uniformly distributed errors with same error probabilities .

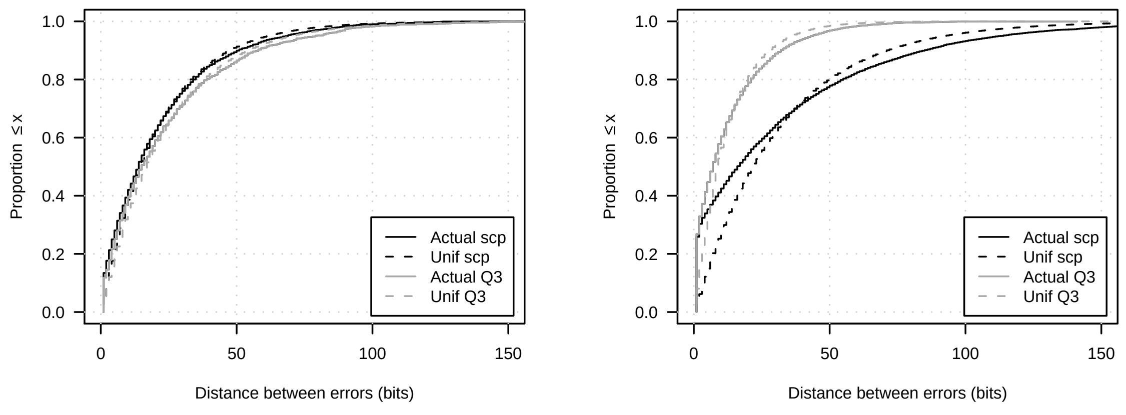


Figure 4.22: Distance between bit errors in bits for sparse encoding (left) and sub-band encoding (right)

Sub-band encoding experiences burstier errors than sparse encoding, and scp produces burstier errors than Q3. With emulated packet loss and reordering errors are even burstier, since lost or reordered packets likely create error bursts and we emulated correlated packet loss and reordering. The emulated network jitter, packet reordering and loss were stationary, and hence the measured error rate was also stationary.

4.5.4 Throughput

We measured the throughput over the channel using the reliable transport protocol developed in Section 3.4. We used sub-band encoding, because for low network jitter its capacity is higher than that of sparse encoding[7]. For all experiments without packet loss and reordering we used the non-deletion technique and for the remaining experiments we used the marker-based technique. We used the same codes for scp and SSH, but different codes for Q3[8]. The code lengths were similar to those used for the TTL channel (see Appendix C.6). The duration of experiments with interactive SSH was increased to one hour to increase the number of transmitted data blocks.

The encoding parameters were manually tuned according to the error rates in the different scenarios with the goal of achieving a very low block corruption rate with FEC alone. As in Section 3.5 we computed the actual throughput assuming a FEC+ARQ scheme with a target block corruption rate of 1^{-9}. Because we used relatively high redundancies for the RS codes, in all experiments the block corruption rate was below 0.5%. In many experiments it was actually zero.

[7]We used model-1.0 and set $l = 5$ ms (as before).

[8]For scp and SSH the error rates are similar, but for Q3 the channel has consistently higher error rates.

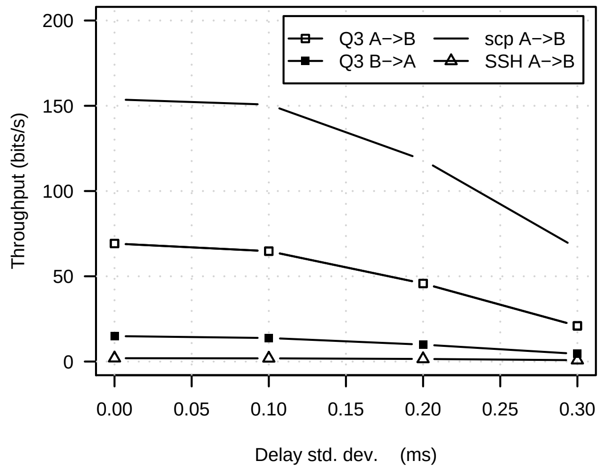


Figure 4.23: Throughput depending on the standard deviation of the network delay for 0% packet loss and reordering

Figure 4.23, 4.24 and 4.25 show the throughput for the different applications and error rates. For scp the data was transferred from Alice to Bob, and the covert channel was encoded in this direction. For Q3 the throughput from Alice to Bob is much larger, as Alice's host running the Q3 client sent one packet every 10–20 ms, but Bob's host running the server only sent one packet every 50 ms [190]. For SSH the covert channel was encoded in the traffic sent by the remote host (target of the SSH session).

The results show that throughput reduces quickly with increasing network jitter, but only moderately for increasing packet reordering. With increasing packet loss the throughput decreases substantially for scp, because not only is more redundancy needed but also the overt packet rate decreases substantially. The throughput is still at least 20 bits per second for overt flows with higher packet rates, such as scp or Q3 client-to-server traffic.

Figure 4.26 shows the percentage of the channel capacity reached for the different applications (averaged over both directions for Q3). We computed the capacity based on the configured packet loss rate, and the substitution error rates measured.

The percentage of the capacity reached is at least 30–40%, except for high packet loss. Unlike for the same experiments with the TTL channel here the impact of packet loss is more severe than the impact of packet reordering, because lost packets not only cause deletions but also substitution errors[9].

The percentage for scp is always the highest. We think the percentage for Q3 is lower because errors are less bursty as shown before, making the RS code less effective. For SSH the percentage is often lower because the error rates are slightly smaller than for scp and hence the code was slightly over-dimensioned. Furthermore, despite the longer

[9]The TTL channel with MED modulation scheme has no additional substitutions caused by deletions.

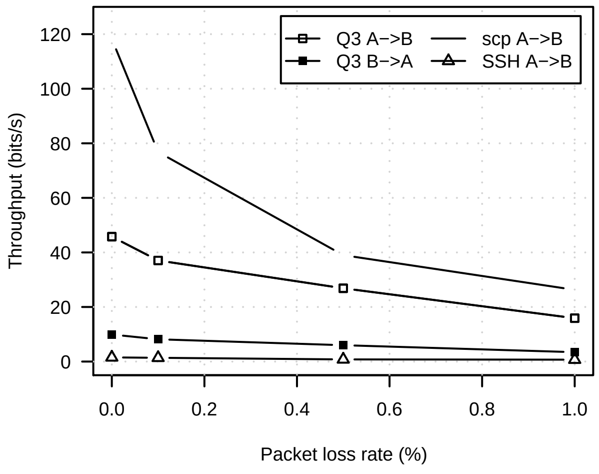


Figure 4.24: Throughput depending on the packet loss rate with 0.2 ms standard deviation of network delay and 0% packet reordering

experiments the results for SSH are still affected by rounding errors due to the small number of blocks transmitted (see Section 3.5.6).

We also examined the variability of the covert bit rate over time (see Appendix C.5). Without packet loss the rate is relatively constant for all applications. With packet loss it is still relatively constant for Q3 and SSH, but for scp it varies significantly.

We outlined in Section 3.5.6 how use of TCP-specific information could improve the capacity of the TTL channel for overt TCP traffic. To a lesser degree similar improvements may be introduced to IPG timing channels. TCP sequence numbers could be used to detect packet loss and possibly also to retransmit covert bits. The latter assumes each bit is uniquely associated to a TCP sequence number and the selection of overt packet pairs is modified, so that if a packet is selected all retransmission of it are also selected.

However, bit errors caused by packet reordering cannot be prevented. Our current implementation already puts packets in the correct order before decoding, but it is impossible to reconstruct the original IPGs.

4.5.5 Buffering delay

We examined the delay introduced to the overt traffic for (semi-)passive channels. We used the buffering delay configuration from the previous experiments and two additional configurations resulting in smaller delays. The network delay was fixed to 25 ms in each direction and packet loss and reordering were set to zero. For the encoding fraction and sub-band size we used the same parameters as before, and we used model-0.5. Figure 4.27 depicts the measured error rate depending on the mean buffering delay.

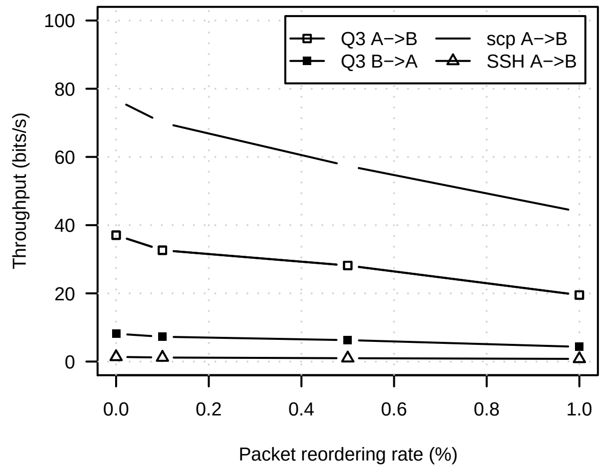


Figure 4.25: Throughput depending on the packet reordering rate with 0.2 ms standard deviation of network delay and 0.1% packet loss rate

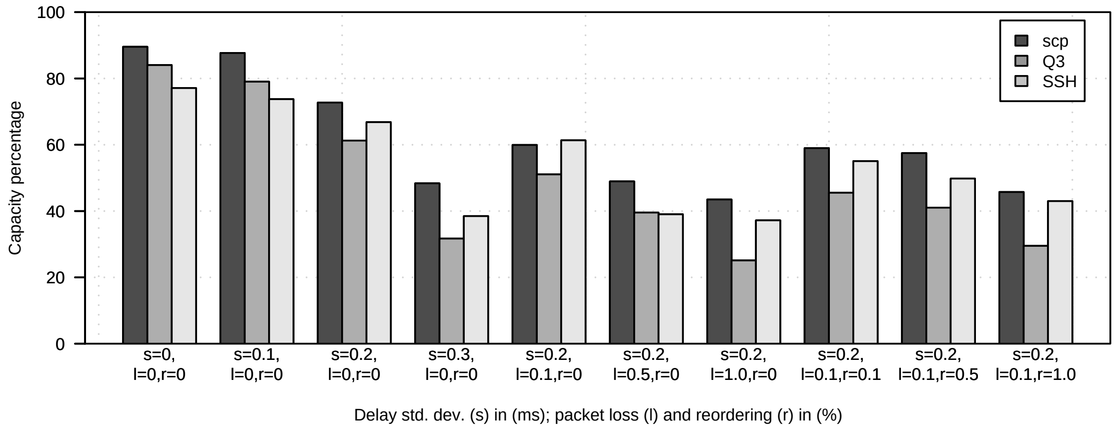


Figure 4.26: Percentage of capacity reached for the different applications and network jitter, packet loss and reordering rates (averaged over both directions for Q3)

The mean buffering delays for the smallest delay setting are similar for sparse and sub-band encoding for all applications. For Q3 the error rate is very similar for all settings. For scp and SSH the error rate increases greatly with smaller buffering delays when using sparse encoding, but for sub-band encoding the increase for scp is more moderate and there is no increase for SSH. Sub-band encoding performs smaller IPG changes than sparse encoding, reducing the chances of buffer underflows. However, if the IPG distribution centres around a single peak, such as for Q3, there is no difference.

For Q3 the smallest delays are about 35 ms (client-to-server) and 70 ms (server-to-client). If the channel is encoded simultaneously in both directions it could get noticed by latency-conscious players, but if encoded in only one direction the added delay is relatively small given that for RTTs of up to 100–150 ms players usually do not notice a

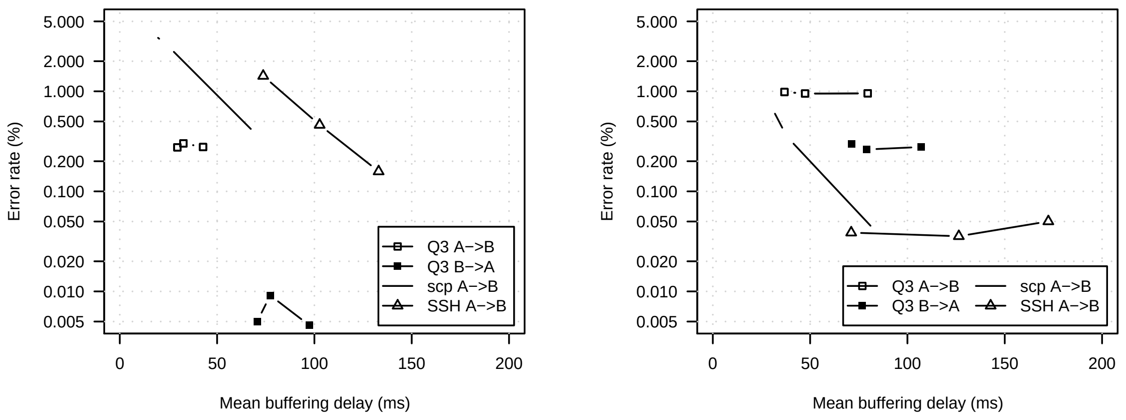


Figure 4.27: Error rate vs. mean buffering delay for sparse encoding (left) and sub-band encoding (right) (note log *y*-axis)

degradation of game-play [202]. For interactive SSH the delay is less than 75 ms (sub-band encoding) and less than 150 ms (sparse encoding), which is tolerable. Because scp was bandwidth-limited, smaller or larger buffering delays do not significantly affect the overt packet rate or throughput, unless there is packet loss.

For model-1.0 the buffering delay is similar for Q3, but significantly increased for scp and SSH. Again, for bandwidth-limited scp the increased delay is only noticeable if there is packet loss, but the now 300 ms delay for SSH might get noticed. However, as said before for semi-passive channels the introduced delay can be large.

4.6 Conclusions

We showed that IPGs of real UDP and TCP traffic are often auto-correlated, which makes previous timing channels proposed in [104, 105] easy to detect. We proposed new improved encoding schemes that are harder to detect, but have either reduced capacity (sparse encoding) or reduced robustness (sub-band encoding). Our new techniques generate the random numbers needed for encoding from the packets themselves. Only this makes the channel usable for UDP-based overt traffic that has no accessible sequence numbers. Our new schemes can be used for active and passive channels.

We created a proof-of-concept implementation and discussed several implementation issues, including how to handle overt traffic dependencies, how to send packets accurately and the choice of hash functions and their input. We also proposed a channel model that allows computing the channel capacity based on the channel errors.

We performed experiments in a testbed with different emulated network conditions, using three applications as overt traffic. Our results show that the new timing channels have low error rates and high capacities for low network jitter. Transmission rates are up to over hundred bits per second. However, the capacity is severely reduced if the network jitter is high. Then it is much smaller than the capacity of the TTL channel.

Sparse encoding has lower error rates since each bit is encoded using the full IPG distribution. However, sub-band encoding provides higher transmission rates because a larger fraction of the IPGs can be used. The wider the IPG distribution of the normal traffic, the more resistant the channel becomes against timing jitter and the greater the robustness improvement of sparse encoding over sub-band encoding. The applications used in our experiments have narrow IPG distributions, but we believe this is typical for many existing applications. However, there are some applications with wider distributions that would make the channel more robust, but the maximum transmission rate would still be relatively low, because such applications also have low packet rates.

For a hybrid FEC+ARQ scheme we measured throughputs of 30–40% or more of the capacity. Even with modest packet loss and reordering the throughput was still over 20 bits per second for two of the applications used, which is much higher than the commonly accepted limit of one bit per second [19].

It is difficult to summarise the comparison of the capacity of the IPG timing channel and the TTL channel, because for both channels the capacity varies significantly based on the noise, and some types of noise affect only one channel but not the other. We make a general comparison assuming TTL noise (1^{-3}) and network jitter ($\sigma = 0.2$ ms), low packet loss and reordering rates (0.1%) and the best encoding schemes under these conditions (MED and sub-band encoding).

Then the TTL channel achieves approximately 0.97 bits per packet and the IPG timing channel achieves approximately 0.78 bits per packet pair. This means the IPG timing channel has 70–80% of the capacity of the TTL channel. This is consistent with the code rates used in the experiments with the reliable transport protocol. The average code rate of the IPG timing channel was about 77% of the code rate of the TTL channel.

The new encoding techniques work well for active channels or semi-passive channels, where the introduced buffering delay is not of concern. Furthermore, even with the currently simple buffering algorithm they are suitable for passive channels in most cases. Only for high-rate TCP flows and higher packet loss rates they cannot be used as the throughput of the overt traffic is severely reduced.

4.6.1 Future work

Our improved timing channels work for both active and passive channels, but we have not analysed the performance of active channels. Also, we have not investigated multi-symbol channels that encode more than one bit per IPG. The analysis could be extended to cover a wider range of applications generating the overt traffic, including some with wider IPG distributions, and to a larger range of network conditions. Furthermore, experiments could be carried out across Internet paths, for example using the PlanetLab overlay network.

There are a number of avenues to further improve passive channels, including the study of bidirectional channels over single TCP flows, developing new passive channels that encode covert data in large IPGs as well, and automatically optimising the location of sub-bands during model creation. The buffering delay management algorithm needed for passive channels should be improved. For example, it could use a prediction of future IPGs of the overt traffic to optimise introduced delay.

Our proof-of-concept implementation is a userspace application, which limits the covert sender's ability to time packets accurately. Future work could investigate accuracy improvements possible with a sender implemented as kernel module, and when using a real-time UDP extension for real-time Linux.

Another interesting area for future research is the study of related timing channels. For example, the TCP timestamps could be used to create a noise-free active channel in TCP traffic, similar to the one in [64] but harder to detect, or the timing of message sequences could be manipulated to create timing channels with various application-layer protocols.

Chapter 5
Covert Channels in Multiplayer Games

In this chapter we develop and analyse a novel indirect covert channel hidden inside multiplayer first person shooter (FPS) online game traffic, called Reliable FPS Covert Channel (RFPSCC). RFPSCC provides reliable data transport for our new covert modulation scheme we call FPS Covert Channel (FPSCC), which itself is unreliable because the covert channel is noisy.

FPSCC is a channel between game clients hidden from game server operators and players. While it is likely that Alice and Bob are players of the game, FPSCC could also be used in scenarios where they are middlemen using the game traffic of other unwitting players. FPSCC encodes covert information in slight variations of character movements intended by human players. We choose variations that have no visible effect on the movements as perceived by human players. Serves usually do not log character movement.

FPSCC has a number of desirable properties for users. FPS games are very common and their network traffic is not suspicious, although it may not be present everywhere. FPSCC is both a broadcast and an indirect channel – one covert sender transmits information to one or more covert receivers using a game server as an intermediary, rather than directly exchanging covert data. Detection of the covert sender does not directly expose the identities of the covert receiver(s) who could be any of the players online at the same time. FPSCC is impractical to eliminate, because it is tied to player movement (an intrinsic function inside the games).

FPSCC could be used for collusion as well as exchanging game-unrelated information unbeknownst to adversaries. Capable adversaries could easily detect overt communication, such as VoIP or traditional and in-game instant messaging. Beyond games, many companies are exploring the use of immersive virtual worlds, such as Second Life [28], for distributed training, collaboration and general business; this opens up the potential for covert ex-filtration of commercially sensitive information via FPSCC-like channels.

First, we provide some background on FPS games and their network protocols. Then we present the design of the modulation scheme (FPSCC). Next, we discuss various sources of bit errors that make FPSCC unreliable. FPSCC experiences bit substitutions as well as bit insertions and deletions (synchronisation errors). Then we present the design of RFPSCC, a unicast bidirectional channel over FPSCC. Finally, we evaluate the throughput of RFPSCC based on a proof-of-concept implementation and compare it with the channel capacity. The channel capacity is estimated based on an information-theoretic model we propose.

Our proof-of-concept implementation is based on the game Quake III Arena (Q3) [188]. We chose Q3 because it runs under Windows and Linux, the source-code is freely available and Q3's general client-server communication architecture is shared by many other FPS games. While Q3 itself is outdated some game modifications based on Q3 were still popular 3–4 years ago, for example Wolfenstein: Enemy Territory [203].

We analyse the throughput and bit error rate of RFPSCC based on a large number of test games with various degrees of network delay and packet loss impacting on the overt traffic. We show that RFPSCC is reliable (no bit errors) and provides throughputs of up to over 15 bits/s (with client-to-server and server-to-client packet rates of approximately 86 and 20 packets per second respectively). RFPSCC has a low throughput, but it is still sufficient for exchanging text messages or chatting.

5.1 Background

5.1.1 First person shooter online games

The publishing model for FPS games, such as Quake III Arena, Unreal Tournament and Counter-Strike Source, makes them intriguing for use as covert channels. FPS games are based on the client-server architecture, and publishers typically release their servers free – relying on Internet Service Providers (ISPs), dedicated game hosting companies and individual enthusiasts to host FPS servers.

FPS game servers typically host from less than 10 to around 30+ players and particularly for popular games there may be tens of thousands of individually operated servers active on the Internet at any given time [190, 204]. Alice and Bob thus have a wide variety of game servers through which to establish legitimate-looking overt traffic flows.

5.1.2 Player movement protocol

Q3 relies on UDP packets to carry information between clients and servers. FPSCC utilises the network traffic that occurs during game play, and ignores traffic associated with server-discovery and initial client connection.

Figure 5.1 illustrates the message flow during game play. *User commands* are sent from client to server once per graphics frame rendered (but no faster than once every 10 ms). *Snapshots* are sent from server to client once every 50 ms (by default). Transmissions of user commands and snapshots are not synchronised.

Figure 5.2 illustrates the movements that may be indicated in each user command. Movement occurs along three axes (x – left/right, y – up/down, z – forward/backward) and

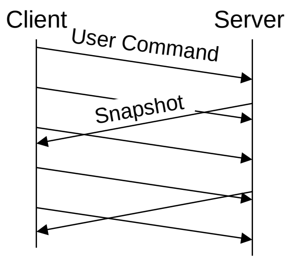


Figure 5.1: Messages exchanged between Q3 client and server

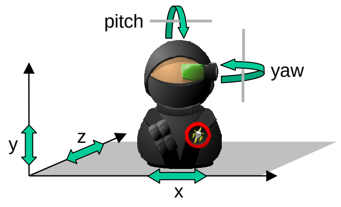


Figure 5.2: Player character movement

change of view angle may be requested along the x- and y-axes (pitch and yaw angles). A user command also indicates movement-unrelated mouse and keyboard button state as well as the selected weapon. To compensate for packet loss, each packet sent by the client contains the current and previous user commands. Client messages also contain 'reliable' commands, such as the "disconnect" command.

Each client receives game world state updates in snapshots. A snapshot contains the server's authoritative belief about the state of the client's player (position, view angles and player-specific events) as well as the state of all other entities potentially visible to the client's player (positions, view angles and events). Entities can be other human player characters, computer-controlled characters (bots) or objects. Entity state updates are not sent for entities that the client's player cannot see; however, not all potentially visible entities are actually visible on the player's screen. This reduces network traffic and mitigates a source of potential client-side cheating (chapter 7, [190]). Server messages also contain 'reliable' commands, such as printing messages on the client's screen.

Q3 uses sequence numbers in both directions to detect loss of packets. If loss occurs, reliable commands are retransmitted. Lost user commands and entity state updates are never retransmitted as they are continuously updated anyway. User commands sent by the client are timestamped, as are player and entity state updates sent by the server. Consequently every update of player state sent by the server can be unambiguously linked to a corresponding previous user command sent by a client.

All user commands and snapshots are delta-encoded to reduce packet size, so a data field is only sent if it has changed, and all messages are compressed using adaptive Huffman encoding [205]. Despite differences in specific details, most FPS game protocols utilise a similar overall design.

Figure 5.3 illustrates the relationship between player position information sent to the server, and the same information received by other clients. Let x_i be client 1's player input for their character's position or view angle along an axis in user command i and let y_j be the position or view angle of client 1's character sent by the server to both clients in snapshot j. We assume x_i and y_i are integer values or the integer part of real values.

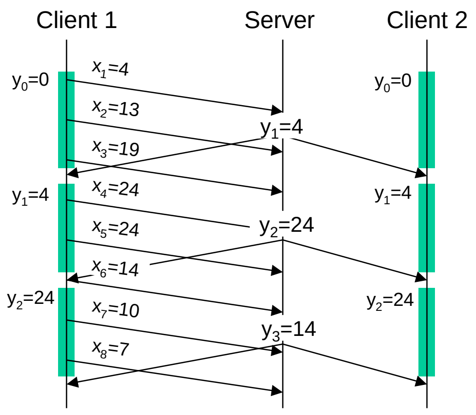


Figure 5.3: Example user input values and server snapshot values

As user commands usually arrive more frequently than snapshots are emitted, each y_j is computed based on the most recently received x_i. Client 2 renders client 1's player on screen based on y_j until it receives y_{j+1}, a period indicated by the boxes.

5.2 FPS covert channel

FPSCC creates a covert channel between two game clients. Alice and Bob may be deliberately built into actual clients or be middlemen manipulating game traffic of unwitting players. Alice encodes covert information by modulating x_i from client 1 with visually imperceptible fluctuations of character movement. Bob decodes covert information from y_j updates arriving in snapshots.

FPSCC is not limited to unidirectional communication, as Bob could send covert information to Alice at the same time. FPSCC aims to avoid detection by either the players controlling the game clients, or by an adversary (Wendy).

5.2.1 Encoding and decoding

We leverage the fact that Q3 encodes more detail in x_i and y_j than is normally observed by a human player. FPSCC modulates players view angle commands for pitch (between -87 and 87 degrees) and yaw (between -180 and 180 degrees), since players' view angles are, with small exceptions discussed later, almost entirely dictated by player input. Other information in user commands is less suitable. Position information may be perturbed by various 'forces' acting on a player's character, making it hard to predict y_j from x_i. Surreptitious manipulation of mouse button or key state is harder to hide from players, and also would result in a very low capacity.

We use *changes* in view angles to encode covert information. To minimise detection, FPSCC only encodes covert information between two snapshots when players are

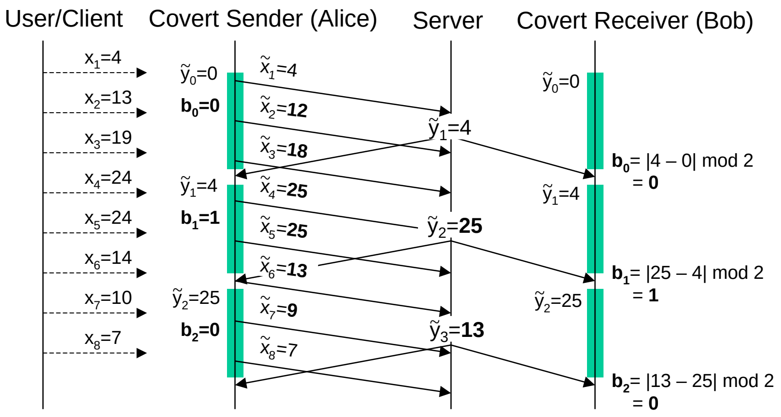


Figure 5.4: Example of covert channel encoding

adjusting their character's view. If a player stops, the covert channel pauses. The covert channel is effectively masked if FPSCC-induced changes are small compared to players own input. Our experience suggests that such a covert channel is unlikely to be noticed by players and would not affect their ability to play. In our experiments none of the human players noticed unusual view angle changes, not even of their own player characters.

Before we discuss the details of the encoding we present an example. Figure 5.4 illustrates the encoding of covert bits using the pitch angle, with a zero start value and the same user input as in Figure 5.3. One bit of covert information is encoded per angle change as per the encoding rule (Equation 5.1). This means an even change signals a logical zero and an odd change signals a logical one. The angle values modified by Alice are shown in bold. The boxes indicate the time periods in which a covert bit is transmitted.

The details of FPSCC's encoding are as follows. Alice encodes N bits of covert information with an integer value of b ($0 \le b \le 2^N - 1$) into each angle change so that:

$$b = \left|\tilde{y}_j - \tilde{y}_{j-1}\right| \bmod 2^N \, , \tag{5.1}$$

where $\tilde{y}_j$ and $\tilde{y}_{j-1}$ are the angle values manipulated by Alice. However, Alice can only indirectly modify y_j by modifying the user input x_i.

As noted previously, the game server computes y_j from the most recently arrived x_i. Q3's asynchronous message transmissions, unpredictable client message rate and variable network delay make it difficult for Alice to predict which x_i will be used by the game server to compute y_j . Therefore, Alice has to encode the same covert bits in all x_i sent between the arrival of y_{j-1} and y_j.

Let the change in user input be $\Delta_i = x_i - x_{i-1}$. When Alice detects an angle change, she starts encoding the next covert bits to be sent b_n in the current and following user commands. Each time a snapshot is received from the server, Alice checks whether the angle value has changed ($\tilde{y}_j \neq \tilde{y}_{j-1}$). If not, Alice continues sending b_n. Otherwise Alice assumes b_n has been successfully transmitted and updates the previous angle value $\tilde{y}_{j-1}$. The next user angle change will cause Alice to start sending bits b_{n+1} and so on.

Even if Alice has not actually sent any bits, an angle change will still cause her to treat b_n as sent. For example, this can happen when Alice could not send anything because there were no user angle changes or user commands, but the server forced an angle change (see Section 5.3). Treating b_n as sent prevents bit deletions and insertions, but bit substitutions may still occur.

Alice encodes covert bits by manipulating x_i, adding a small δ_i:

$$\tilde{x}_i = x_i + \delta_i \,. \tag{5.2}$$

If $\Delta_i \neq 0$ Alice encodes b by selecting δ_i such that:

$$b = \left|\tilde{x}_i - \tilde{y}_{j-1}\right| \bmod 2^N \,. \tag{5.3}$$

From equation 5.2 and equation 5.3 follows:

$$\delta_i = \begin{cases} b - \left(x_i - \tilde{y}_{j-1}\right) \bmod 2^N \,, & x_i - \tilde{y}_{j-1} \geq 0 \\ -b - \left(x_i - \tilde{y}_{j-1}\right) \bmod 2^N \,, & x_i - \tilde{y}_{j-1} < 0 \end{cases} \,. \tag{5.4}$$

However, Alice must avoid completely negating the angle change. If $\left(x_i - \tilde{y}_{j-1}\right) + \delta_i$ equals zero she needs to modify δ_i accordingly:

$$\delta_i = \begin{cases} 2^N + \delta_i \,, & x_i - \tilde{y}_{j-1} \geq 0 \\ -2^N + \delta_i \,, & x_i - \tilde{y}_{j-1} < 0 \end{cases} \,. \tag{5.5}$$

Alice minimises the angle changes she introduces and thereby increases FPSCC's stealth by modifying the angle computed using equations 5.4 and 5.5 as follows:

$$\tilde{x}_i = \begin{cases} \lfloor \tilde{x}_i \rfloor \,, & x_i - \tilde{y}_{j-1} \geq 0, \tilde{x}_i > 0 \\ \lfloor \tilde{x}_i - 1 \rfloor \,, & x_i - \tilde{y}_{j-1} \geq 0, \tilde{x}_i \leq 0 \\ \lfloor \tilde{x}_i + 1 \rfloor \,, & x_i - \tilde{y}_{j-1} < 0, \tilde{x}_i > 0 \\ \lfloor \tilde{x}_i \rfloor \,, & x_i - \tilde{y}_{j-1} < 0, \tilde{x}_i \leq 0 \end{cases} \,. \tag{5.6}$$

The next snapshot value $\tilde{y}_j$ will be based on one of the $\tilde{x}_i$ arrived at the server between snapshot $j-1$ and j, and possible noise on the channel n_j. Bob decodes the covert bit(s) similar to equation 5.3:

$$\hat{b} = \left|\tilde{y}_j - \tilde{y}_{j-1} + n_j\right| \bmod 2^N \,. \tag{5.7}$$

Bob is not able to distinguish normal angle changes without covert bits encoded from changes with encoded covert bits. If Alice continuously transmits bits there is no problem, but she may not have data to send all the time. In the latter case Alice either transmits dummy bits or does not encode bits into angle changes. In both cases Bob identifies blocks of valid data based on higher-layer semantics.

In general the problem is solved if Alice sends data in blocks of bytes with headers containing a block identifier (preamble). The preamble identifies the start of a block and needs to be 'unique' so it can be distinguished from data and pseudo-random bits caused by user angle changes when Alice is not sending or from dummy bits. If the channel carries text characters a simple protocol works as follows. Alice transmits continuously, sending dummy zero bytes if necessary. Bob treats zero bytes as terminators of messages.

Byte and block synchronisation between Alice and Bob are discussed in more detail in Section 5.4. Another question is how Alice and Bob can identify and authenticate each other. This is discussed in Section 5.2.3.

5.2.2 Number of encodable bits

In the simplest case Alice sends one bit per angle change ($N = 1$). However, N could be selected based on the user input: the larger the user's angle change, the larger Alice's modification can be without compromising the stealth of the channel, such as by materially impacting on game play or creating visible anomalies.

We define the dimensionless variable L as the limit of the ratio of the absolute values of Alice's modification and the total angle change (where $0 < L < 1$):

$$\frac{\max\left(|\delta_i(b, N_i)|\right)}{\max\left(|\delta_i(b, N_i)|\right) + |\Delta_i|} \leq L \,. \tag{5.8}$$

We also assume there exists an absolute maximum $\delta_{\max}$ limiting Alice's modification regardless of Δ_i. Then the number of encoded bits N_i is chosen to be (see Appendix D.1):

$$N_i \leq \min\left(\left\lfloor \log_2(\delta_{\max} + 1)\right\rfloor, \left\lfloor \log_2\left(\tfrac{1 + L\cdot(|\Delta_i| - 1)}{1 - L}\right)\right\rfloor\right). \tag{5.9}$$

Figure 5.5 illustrates the number of encodable bit over the user change Δ_i for different values of $\delta_{\max}$ and L. For different $\delta_{\max}$ the lines are slightly offset for improved visibility.

5.2.3 Broadcast and unicast channels

As previously noted, Bob only receives snapshots containing $\tilde{y}_j$ updates relating to Alice if Alice's in-game character is potentially visible to Bob inside the game environment. Consequently, there are two transmission modes open to Alice – *unicast* and *broadcast*.

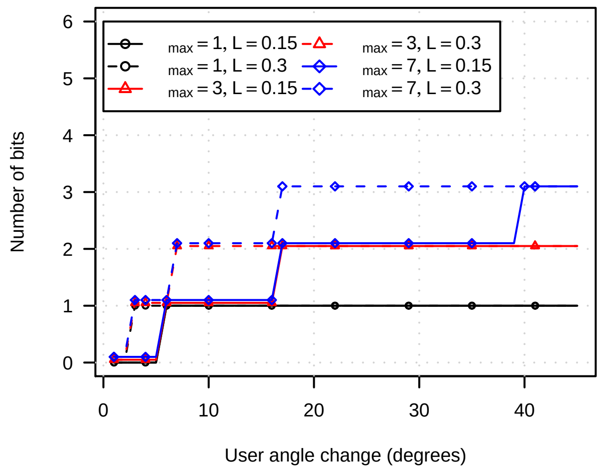


Figure 5.5: Number of bits encodable vs. user angle change for different δ_{max}, L

If Alice knows the in-game identity of Bob's client then she can chose to only send covert data when Bob's character is known to be in range, as determined from the server snapshots. This is unicast mode, and the covert channel may pause from time to time. However, if Alice has no idea on which client Bob resides, she simply transmits covert data continuously. This is broadcast mode, where the covert channel may experience significant periods of lost bits.

Regardless of Alice's transmission mode, Bob is not required to know in advance on which game client Alice resides. Bob can simply attempt to decode covert information from $\tilde{y}_j$ updates relating to every player. Any stream of $\tilde{y}_j$ updates that generates 'meaningful' covert information can be presumed to come from Alice. "Meaningful" could mean the existence of pre-defined bit sequences as used in [122] or data structures previously agreed upon by Alice and Bob. However, if Alice's client's identity (e.g. player name) is known, Bob can focus on decoding $\tilde{y}_j$ updates from that specific player.

FPSCC in multicast mode allows multiple instances of Bob, each associated with a different game client. Each Bob decodes part or all of the covert channel's data stream as their associated players cross paths with Alice inside the game's virtual world.

5.2.4 Simultaneous yaw and pitch encoding

FPSCC supports encoding bits in both pitch and yaw simultaneously. In our scheme Bob always decodes the bits in fixed order, the first N_p bits from pitch (b_n) and the second N_y bits from yaw (b_{n+1}). But Alice has to encode the bits in the order the user changes angles between snapshots. For example, if a yaw change in user command i is followed by a pitch change in user command $i+1$, Alice encodes b_n in yaw and b_{n+1} in pitch. This is a problem, because in this case FPSCC would actually swap bits b_n and b_{n+1}.

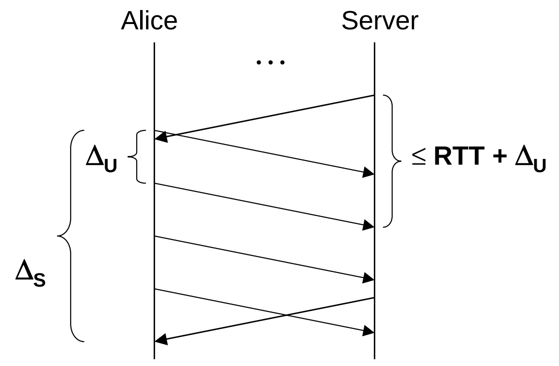


Figure 5.6: Round trip time limit of FPSCC

To prevent this, Alice needs to swap bits b_n and b_{n+1} as soon as she receives user command $i+1$. However, it may still happen that after Alice has swapped the bits user command $i+1$ arrives after the server has already created the next snapshot and effectively user command i is used. If Alice discovers this from snapshot j, she swaps the bits again (b_n was sent encoded in yaw and b_{n+1} remains to be sent).

To avoid bit errors Alice then encodes b_{n+1} in pitch in the user command following snapshot j to 'override' the previously encoded b_n. Since the previous pitch change was based on user movement, this does not violate the "no encoding if no user change" policy.

5.2.5 Impact of round trip time

Figure 5.4 assumes that the RTT between Alice and the server, plus the time between two user commands, is smaller than the time between two snapshots. Otherwise, during encoding Alice would not know the actual value of $\tilde{y}_{j-1}$ and hence could not compute the correct δ_i. However, there is no limit for the RTT between Bob and the server.

Let Δ_u be the time between user commands and Δ_s the time between snapshots. Then, the maximum tolerable RTT for Alice is (see Figure 5.6):

$$\mathrm{RTT} \leq \Delta_\mathrm{s} - \Delta_\mathrm{u} \,. \qquad (5.10)$$

FPS games are sensitive to latency and players would usually only choose servers with RTTs of less than 100–150 ms [206, 202]. Given that server updates usually occur every 50 ms, and client messages often every 10 ms, FPSCC would be limited to situations where RTT $\leq$ 40 ms. FPSCC works over larger RTTs at the cost of reducing the capacity. The bits b_n are now sent in m server snapshots, where $m \geq 2$ and $m \cdot \Delta_\mathrm{s} - \Delta_\mathrm{u} \geq \mathrm{RTT}$. The server-time timestamps t_s are used to control which snapshots contain covert bits. Alice and Bob only encode and decode from snapshots where $t_\mathrm{s} \bmod m = 0$.

To improve performance Alice could send bits in overlapping time periods between m snapshots (pipelining). Alice encodes covert bits every snapshot based on the current angle and the old angle $\tilde{y}_{j-m}$ and Bob decodes accordingly. Alice and Bob need to buffer the last m angle values. However, unfortunately this method does not work in combination with the technique for encoding simultaneously into both angles described above.

Alternatively, Alice could predict the angle of the next snapshot and use the predicted angle for encoding. The accuracy of the prediction depends on the jitter inherent in the timing of client and server messages and the network jitter. We leave the design of a more efficient approach as future work.

5.2.6 Deployment considerations

Cheat protection built into multiplayer FPS games makes deployment of covert channels challenging. Client software and data files are checked for integrity and the memory of the client machine is searched for signatures of known cheats. Games typically encrypt their network protocol to protect against proxy-based cheats.

It is straightforward for a player to act as Bob. FPS clients usually allow recording of demo files, which contain all the unencrypted entity state updates seen during the recorded game sequence. Bob simply records a ‘demo’ and decodes the covert data after the game. However, if Bob is not a player he has to intercept the network traffic like Alice.

Alice needs to modify the protocol data during the game. We implemented FPSCC using a proxy-based approach, because Q3 is open source and hence the encryption algorithm is known. However, even if an encryption algorithm is not publicly known it may be possible to crack it as described in [207]. Proxies cannot be detected by current anti-cheat software integrated in games.

Alice could also be deployed on the client as a client-side modification (called “mod”). However, many public servers do not accept modified clients. Alice can still be deployed on the client like other client-side cheating tools. These tools work without requiring a modification of the game client and can only be detected if the anti-cheat software knows their signature. Even if Alice’s signature became known, it could be modified easily to evade the signature detection. Currently, no method exists for reliably detecting unknown client-side cheats.

5.2.7 Implementation considerations

The Q3 network protocol is encrypted. Our implementation of Alice accesses traffic in both directions to acquire and update the Q3 encryption keys, decrypt user commands, perform the covert channel modulation, and re-encrypt the modified user commands. Bob passively decrypts packets in both directions and decodes the covert channel from the snapshots. Our prototype of FPSCC is based on CCHEF (described in Appendix A).

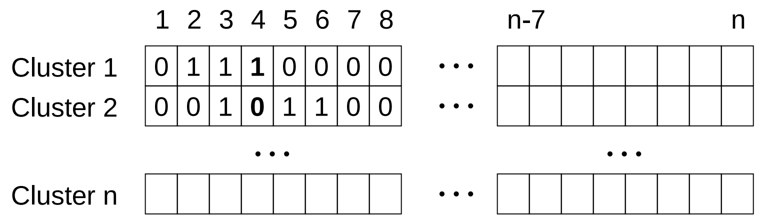


Figure 5.7: Visibility matrix contained in Q3 map files defines potential visibility between different parts (clusters) of the map

Our implementation properly handles the various encodings of view angles in different Q3 messages. In user commands angles are encoded as signed 16-bit integer ($\frac{\alpha}{360} \cdot 65535$), angles in player state are encoded as floating-point numbers and angles in entity state are encoded as the integer parts of the floating-point numbers. The integer-floating-point conversion introduces rounding errors and Alice ensures that these do not cause bit errors. Our implementation also handles wrapping of the yaw angle, which wraps around from -180 degrees to 180 degrees and vice versa.

Our implementation supports map changes and client disconnects. During a map change or when a client is disconnected the channel pauses. But it resumes seamlessly when the new map has been loaded or the client has been reconnected.

5.3 Sources of bit errors

5.3.1 Visibility

The Q3 server does not send the state of an entity to a player, if the entity is not potentially visible to the player. Whether an entity is potentially visible is decided based on the Potentially Visible Set (PVS) information contained in map files and the actual positions of the player and the entity on a map.

PVS information is computed during map creation. The map editor divides a map into n clusters and for each cluster computes which other clusters are potentially visible [208, 209]. The PVS is stored as $n \times n$ bit matrix in the Q3 map file. A cluster i is potentially visible from a cluster j when the i-th bit in the j-th row of the matrix is set. For example, given the PVS shown in Figure 5.7, cluster 4 is visible from cluster 1, but it is not visible from cluster 2. To determine what entities are potentially visible to the player in the current snapshot the server checks if any of the clusters a bounding box around the entity touches is visible from the cluster the player's character is in.

The PVS does not define what exactly is visible on a player's screen. This is determined by the rendering process on the client. The PVS information is used to cull the number of objects before the rendering in order to reduce the rendering time. It is also

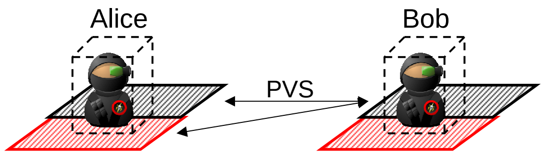


Figure 5.8: An example of asymmetric visibility in Q3, where Alice can see Bob, but Bob cannot see Alice.

used to reduce network traffic and mitigate cheating by only sending entity state updates for entities a player can potentially see (chapter 7, [190]).

In Q3 potential visibility is asymmetric. This means that if Alice can see Bob, Bob cannot necessarily see Alice. There are two reasons for the asymmetry. Firstly, the PVS matrix in the map file may contain asymmetries. We presume this is because of the way the complex PVS algorithm works. Secondly, the way the server determines visibility for the purpose of entity state updates in snapshots also leads to asymmetries. The server only checks visibility between the *one* cluster given by the player character's position with *all* clusters the bounding box around the other player touches.

Let C_{A} and C_{B} be the set of clusters a bounding box around Alice's and Bob's player character touches, and let c_{A} and c_{B} be the clusters Alice and Bob are in. It can happen that Alice sees Bob because one cluster in C_{B} is visible from cluster c_{A}, but Bob cannot see Alice because no cluster in C_{A} is visible from c_{B}. For example, in Figure 5.8 Alice's and Bob's characters are in the front clusters but their bounding boxes touch the front and back clusters. Visibility is asymmetric with the given PVS (Alice can see Bob, but Bob cannot see Alice).

For FPSCC the asymmetric state exchange is problematic, because it does cause bit insertions and deletions. In Section 5.4 we describe RFPSCC, which synchronises the transmission and reception of bits. Furthermore, we developed a modified version of Q3 with symmetric visibility (see Appendix D.2), which allowed us to analyse the throughput of FPSCC with symmetric state exchange and compare it against RFPSCC (see Section 5.5). Since other FPS games may use a symmetric PVS our modified Q3 implementation also demonstrates the applicability of FPSCC to these games.

5.3.2 Player death and teleportation

When players die they respawn after a short time at one of several map locations. Respawning changes the player's position as well as the view angles. This server-enforced angle change can cause bit errors if the dead player respawns within the potentially visible range of the other player, since the resulting angles depend solely on the server. But bit errors can be prevented as players can detect their own death and the death of other players.

If Alice has died she stops encoding covert data until fully respawned, but still keeps track of her own angle changes. If Bob is also sending, Alice continues to decode covert data from him. Bob continues to encode covert data, but stops decoding from Alice until she has fully respawned. However, he continuously tracks Alice's angle changes.

This approach presupposes that the "Alice is dead" and "Alice is alive again" signals are synchronised between Alice and Bob. This may not be the case when Bob loses visibility of Alice after her death but before her respawn. Then Bob may never receive the "Alice is alive again" signal and would be 'out of sync' with Alice. RFPSCC enforces a re-synchronisation of the channel after one or both players died.

Players may enter teleportation devices (e.g. portals), which will teleport them to another place on the map. Teleportation may not only change the player's position but also the view angles. This is effectively the same as players' respawning and is handled in the same way.

5.3.3 Angle clamping and movers

The Q3 server clamps the pitch angle ($\tilde{y}_j$) between a minimum of -87 and a maximum of 87 degrees in snapshots, regardless of the angle indicated in a client's previous user command ($\tilde{x}_i$). This can cause bit substitutions in the snapshot where the angle is clamped and bit deletions in further snapshots while the angle remains clamped.

To avoid these errors, Alice and Bob do not encode or decode bits in snapshots where they discover the angle is clamped until the angle plus Alice's modification is within the allowed range again. In practice players usually do not look straight up or down, so pitch clamping is uncommon (see Section 5.5.1).

A mover is a moving platform that player characters can stand on. If the mover rotates, view angles change not only based on the player's input, but also depend on the rotation of the mover. Therefore, bit substitutions may occur. RFPSCC prevents bit errors by enforcing a re-synchronisation of the channel if Alice detects that she did not sent the bits she intended to send based on the actual angles value in the snapshot.

5.3.4 Packet loss and reordering

As Q3 uses UDP, user commands and snapshots can be reordered or lost in the network. We assume that Alice and Bob do not buffer any messages to avoid additional latency, so reordered messages are effectively lost. We first discuss the effects of loss on user commands and then on snapshots.

User commands are highly redundant as several are sent between two snapshots. If no user commands reach the server between snapshots, Alice is simply not able to send bits. However, if some user commands arrive it is crucial that at least one of them has covert bits encoded based on the angles from the previous snapshot (see Section 5.2.5). Otherwise, substitution errors may occur. Lost user commands never cause bit deletions or insertions, because Alice always knows whether any bits were sent from the snapshots.

Loss of snapshots is worse than loss of user commands. If the same snapshot is lost for Alice and Bob FPSCC remains unaffected. But a snapshot that is lost for either Alice or Bob causes bit deletions and insertions in the lost snapshot as well as possible substitution errors in the following snapshot. Using Q3's sequence numbers Alice and Bob can detect lost snapshots and act accordingly (see Section 5.4.3).

5.4 Reliable data transport

We now develop Reliable FPSCC (RFPSCC) – a protocol to cope with the various bit errors. Our novel scheme, tailored to the characteristics of FPSCC's overt channel, adds bit synchronisation, a framing layer to provide byte and frame synchronisation, and a transport layer to provide encryption. We focus on using FPSCC in unicast mode – an equivalent for broadcast FPSCC is left for future work.

5.4.1 Bit synchronisation

The basic idea is that Alice explicitly lets Bob know whether she can see him or not and in the same way Bob informs Alice. This requires the use of two special channel symbols. If Alice sends an UNSYNC symbol to Bob, she indicates that she cannot see Bob. If Alice sends a SYNC symbol to Bob, she indicates that she can see Bob. A drawback of using two special symbols is the increased amplitude of the induced angle changes in order to have $2^N + 2$ symbols. Alternatively, Alice could send special bit patterns, but this would decrease the throughput.

Our scheme works even in the presence of substitution errors, because Alice always determines whether she has sent (UN)SYNC from the snapshots rather than what she *intended* to send. Since RFPSCC is a bidirectional channel we extend our notion of covert sender and receiver to covert peers.

Alice and Bob implement the state machine shown in Figure 5.9. Initially Alice and Bob are in IDLE state. A peer in IDLE state sends UNSYNCs. When a peer in IDLE state sees the other peer it goes into LISTEN state. In this state a peer sends SYNCs. Only when both peers send a SYNC to each other in the same snapshot the channel's state changes to OPEN, because only then both peers can be sure that they can see each other. Furthermore, Alice and Bob only enter the OPEN state if both players are alive. Covert data is only exchanged when the channel is OPEN. An OPEN channel goes back

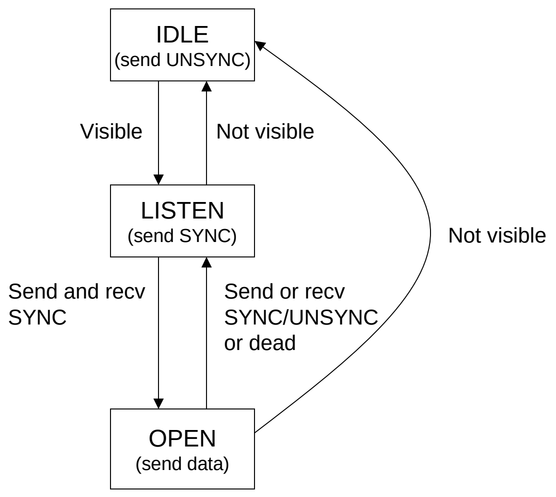


Figure 5.9: Bit synchronisation state machine

to LISTEN state if an UNSYNC or SYNC is received, or back into IDLE state if visibility is lost. The time period the channel is OPEN is called a *transmission period*.

There are two causes of teardown. Firstly, if one or both peers' players die, both peers go into LISTEN state. This prevents the case that a dead player loses visibility and is then unable to send an UNSYNC. Dead players cannot send anything because they cannot move. Q3 sends player-death signals synchronously to all players, so no further synchronisation problem arises.

Secondly, if one peer loses visibility to the other peer, it changes into IDLE state. From the next snapshot on it then sends UNSYNCs. The other peer either loses visibility or receives an UNSYNC and then also ends the transmission period. Whichever peer lost visibility last has potentially sent bits the other peer could not receive. To avoid bit deletions these bits need to be re-sent. The problem is to determine the exact number of bits. If a peer receives an UNSYNC it knows that the other peer lost visibility in the previous snapshot and can re-send any bits sent in this and the previous snapshot[1].

However, there is still the case that neither peer receives an UNSYNC, making it impossible to re-send the correct number of bits. This happens when one peer loses visibility and the other peer loses visibility in the following snapshot. We solve this problem by involving the framing layer (see below). Nevertheless, the maximum number of bit deletions is limited to $4N$ (both angles changed in this and the previous snapshot).

RFPSCC also forces the end of a transmission period if a peer detects that it did not send the bits intended to be sent based on the actual angle values in the next snapshot. This prevents bit errors from movers. To avoid pitch-clamping errors, a peer does not encode or decode bits in snapshots where the angle plus FPSCC's modifications is clamped.

[1]In the rare case that there is no user angle change in the snapshot following the visibility loss, RFPSCC enforces an angle change.

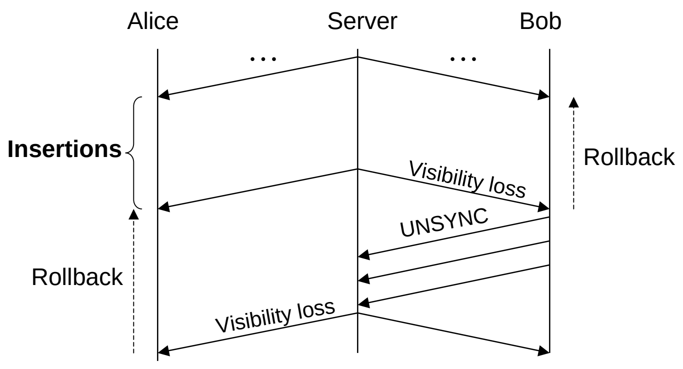


Figure 5.10: Rollback of bits sent at the end of frames

To support larger RTTs FPSCC only encodes bits in every m-th snapshot where m is selected based on the maximum RTT and the time between user commands (see Section 5.2.5). Even when clients normally send user commands every 10 ms, occasionally there can be larger gaps, which cause bit errors if the maximum tolerable RTT is exceeded. To avoid these errors a peer ends a transmission period when it detects that the gap between two consecutive user commands is larger than a configurable threshold. It sends an UNSYNC to the other peer, which then also ends the transmission period.

5.4.2 Framing and transport

The main tasks of the framing layer are to solve the bit synchronisation problem and to identify blocks of bytes in the bit stream (byte synchronisation). There exist a number of framing techniques, for example fixed or variable block length, or start of frame sequence plus bit stuffing (e.g. High-level Data Link Control – HDLC), or CRC (e.g. Asynchronous Transfer Mode – ATM).

RFPSCC's framer uses lower-layer information for framing. All the bits in a transmission period are treated as one frame. Any incomplete bytes at the end of frames are discarded by the receiver and re-sent by the sender. The teardown bit synchronisation problem is solved as follows.

Both peers re-send all bits sent in the last two snapshots before teardown, resulting in at most $4N$ bit insertions but no bit deletions (see Figure 5.10). For $N \leq 2$ at most one byte is inserted. To prevent byte insertions, each peer sends the parity of the length of the previous frame (odd or even) at the start of the following frame and the decoding is delayed by one frame. If the actual parity of a frame differs from the parity indicated by the sender, the receiver discards the last byte before passing the data to the transport layer.

Our scheme supports larger N by aligning frame sizes on multiples of eight bits. For example, if frames sizes are aligned to 16 bits (word-aligned) up to 16 inserted bits can be tolerated and hence N can be up to four. Frame parity is then computed based on the number of words.

Table 5.1: Average per-frame overhead for the different framing techniques

Scheme	Overhead (bits/frame)
Block length	0
HDLC	$16 + 0.0161 \cdot (s+8)$
CRC	40
RFPSCC	4.5 (byte-aligned) 8.5 (word-aligned)

The RFPSCC framing scheme has low overhead. To ensure byte synchronisation, any bits of incomplete bytes at the end of frames are retransmitted. Assuming a uniform distribution of the number of extra bits at the end of frames, the mean overhead is:

$$\overline{o} = \frac{1}{w} \sum_{i=0}^{w-1} i \,, \tag{5.11}$$

where w is the number of bits the frame is aligned on. For example, $w = 8$ for byte-aligned frames and $w = 16$ for word-aligned frames. This means the mean overhead is 3.5 bits per frame if frames are byte-aligned or 7.5 bits per frame if frames are word-aligned. Since the number of substitution errors is effectively zero (see Section 5.5), only a single parity bit per frame is required.

In comparison the overhead of HDLC is an 8-bit preamble plus a number of stuffed bits per frame, whereas the overhead for CRC-based framing is 32 bits. Furthermore, both HDLC and CRC framing need a sequence number to detect which frames arrived. This adds to the overhead and also increases the probability of corrupted frames. Simple block length framing has zero overhead, but it can only be used for unicast transmissions in the absence of synchronisation errors (symmetric visibility and zero packet loss).

Table 5.1 summarises the average overhead in bits per frame for all framing techniques. In case of HDLC the overhead depends on the size of the payload data in bits s and the rate of stuffed bits $R = 0.0161$ [11]. We assumed fixed-size blocks (no frame length field), 8-bit sequence numbers are used with HDLC and CRC framing, and the payload data is uniformly random distributed.

There are no substitution or synchronisation errors above the framing layer. This means the transport layer can use fixed block sizes and block ciphers for encryption, and further error detection or correction techniques are not necessary.

5.4.3 Packet loss

Only the loss of snapshots affects bit synchronisation. To avoid synchronisation errors we extend our scheme so that transmission periods also end if one or more snapshots

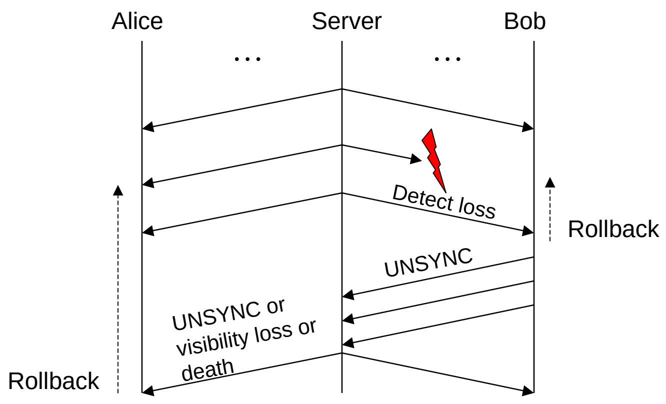


Figure 5.11: Rollback of bits sent at the end of frames after loss of snapshots

were lost that are used for encoding and decoding (depending on the RTT only every m-th snapshot is used). Each peer detects lost snapshots by tracking the Q3 sequence numbers.

For example, if Bob is in state OPEN and detects that snapshots were lost, he goes into LISTEN state. In the next snapshot he will then send an UNSYNC to Alice who will then also change to LISTEN state. Figure 5.11 illustrates this sequence of events. This teardown sequence is basically the same as described in Section 5.4.1.

The only difference is that more bits need to be re-sent. To avoid bit deletions in any possible sequence of events Alice needs to re-send all bits sent in the snapshot(s) lost for Bob and the two snapshots afterwards. However, Bob has no way of indicating to Alice how many snapshots were lost. Hence the number of bits to be re-sent is fixed during operation and must be configured according to the maximum possible number of consecutive lost snapshots $l_{\max}$ (maximum loss burst). The maximum number of inserted bits is then $(2 + l_{\max}) \cdot 2N$.

In order to cope with more than eight inserted bits, frame sizes must be aligned to multiples of eight bits as described previously. For example, if $N = 2$ and bursts of up to two lost snapshots shall be supported the maximum number of inserted bits at the end of frames is 16. Then word-aligned frames guarantee that all bit insertions are corrected.

Lost user commands can cause bit substitutions. These are prevented because a peer also ends a transmission period when it detects that the bits that were actually sent are not equal to the bits that should have been sent (as described in Section 5.4.1).

5.4.4 Alternatives

Because of the asymmetric state exchange FPSCC has a significant number of deletions and insertions that occur in bursts. In our experiments we measured 3–4% deletions and 1–2% insertions. In reality rates will likely vary depending on the map, the number and behaviour of players etc.

Some coding schemes have been developed to deal with deletion and insertion channels. However, as Mitzenmacher points out in a recent paper, "[...] the problem of coding for the deletion channel and other channels with synchronization errors [...] remains a largely unstudied area." and "[...] most of the work in this area remains fairly ad hoc." [20]. Many schemes we examined have limitations which make them unsuitable for FPSCC, such as they can handle only small insertion/deletion rates of less than 1%, they can handle either insertions or deletions but not both, or they assume insertions and deletions are not bursty. Also, implementations are usually not readily available.

Furthermore, as demonstrated in previous chapters, coding schemes typically require careful parameter tuning in order to provide zero error rates with minimum overhead. But this optimum is difficult to achieve given FPSCC's variable error rates. RFPSCC performs well for varying error rates without much tuning.

5.5 Throughput

We analyse the throughput of RFPSCC depending on various factors such as the number of bits encoded per angle change, the network delay and packet loss.

5.5.1 Testbed setup

Our experiments were a mix of tests in a controlled testbed and across real Internet paths, with test machines consisting of two covert game clients, a normal Q3 game client and a Q3 game server[2]. The covert game clients ran the Q3 client and CCHEF with RFPSCC module. To avoid bias, the covert data was uniform random (same probability of one and zero bits), as one would expect if Alice and Bob encrypted their data.

For delay and loss emulation we used Linux Netem [189]. The Linux kernel's tick frequency was set to 1 kHz in order to emulate delays accurate to ±1 ms. We verified that Netem is accurate prior to running the experiments (see Appendix F).

We emulated RTTs of 25 ms (close server and fast network access), 75 ms (close server and typical network access) and 125 ms (further away server and typical network access). We chose 125 ms to be the maximum delay as previous research showed that players aim for a maximum RTT of 100–150 ms and higher RTTs affect their performance [206, 202]. We used constant symmetric delays, since only the maximum RTT matters.

We emulated loss rates of 0%, 0.5% and 1% in each direction. For RFPSCC the limiting factor is not the loss rate, but the maximum number of consecutive snapshots

[2]The two covert game clients had 2.4 GHz Celeron CPUs, and GeForce 7300GT and GeForce MX 440 graphics cards. The normal game client had a 2.8 GHz Pentium CPU and an Intel on-board graphics card. All clients and the server ran ioquake3 1.35 on Linux 2.6.18.

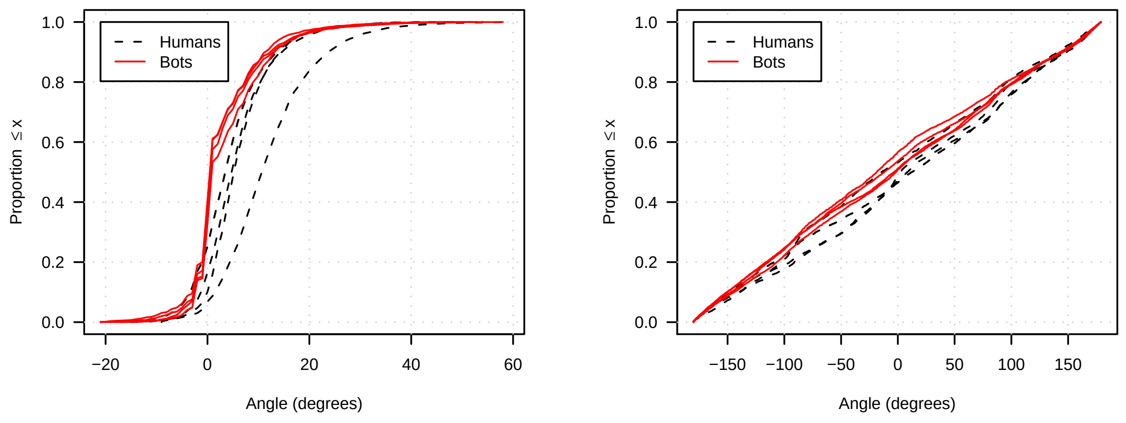


Figure 5.12: Angle distributions for four different human players and four different bots with same configuration for pitch (left) and yaw (right)

lost. We assumed a maximum loss burst of two snapshots for 0.5% and three snapshots for 1% loss rate, meaning a loss burst lasted less than 150 ms and 200 ms respectively. Previous studies showed that loss in the Internet is often below 1% [183, 184] and our maximum burst length is only slightly less than the median reported in [184]. This is consistent with our own Internet measurements showing that loss rates are far below 1% given high-speed broadband access (see below). Furthermore, while Q3 tolerates some loss, it needs to be reasonably low for good game-play [202, 210].

Long measurements with human players are problematic, as exhaustion or change in playing style over time possibly introduces bias in the results. Therefore we used *client-side* bots as players that behave consistently and never get tired [211][3]. We configured the bots to play as human-like as possible (e.g. limited their speed and limited their vision to 180 degrees). However, the bots were probably still faster than the average human player.

We performed a limited number of experiments with nine human players in order to compare throughput and angle distributions of humans and bots. Figure 5.12 compares the angle distributions of human players and bots without FPSCC. Overall the distributions of the bots are similar to that of human players showing uniformly distributed yaw and s-shape pitch distributions. However, the variability between different human players is higher than the variability between different bots with the same configuration.

The pitch distribution of the bots has some ‘steps’ compared to the smooth distribution for humans. This is also noticeable on screen as abrupt pitch movements. We do not know the reason for this, as we do not have access to the source code or specifications of the bot. The pitch distributions also illustrate that pitch clamping never occurred on the flat map we used; pitch angles were always between −20 and 60 degrees.

[3]Q3’s built-in bots cannot be used as they are part of the server.

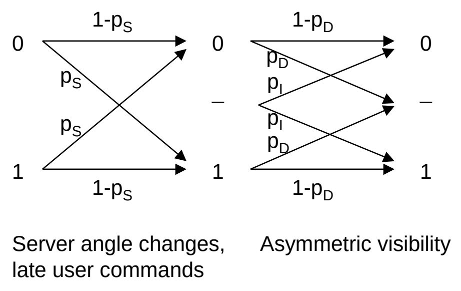


Figure 5.13: Channel model for FPSCC

The angle change rates of bots and humans are relatively similar for yaw: 8.6 changes per second for bots compared to 9.3 changes per second for humans. But for pitch there is a larger difference: 1.7 changes per second for bots compared to 6.0 changes per second for humans. On the flat map bots do not need to change pitch very often, but randomness in mouse movements causes more pitch changes for human players.

The bots produce less angle changes per second. The throughput of RFPSCC is likely to be up to 50% higher for human players if one bit is encoded per angle change, which is consistent with the results (see Section 5.5.4). However, the bots allow us to qualitatively compare the influence of different parameters based on large datasets.

In all experiments we used the map *q3dm1*. The map was restarted every 10 minutes, because deathmatch games typically run for only 10–15 minutes.

5.5.2 Channel capacity

We propose an information-theoretic channel model for FPSCC to estimate the channel capacity. Later we compare the measured throughputs with the channel capacity.

The output of the channel only depends on the input and the errors, but not on previous inputs. The channel has substitution errors as well as bit insertions and deletions. Hence we model the channel as memoryless combined deletion/insertion/substitution channel with error rates p_D, p_I and p_S (see Figure 5.13).

Deletion/insertion/substitution channels have not been very well studied [20]. Their exact capacity is not known but Gallager proved a lower bound [171, 173]:

$$C \geq 1 - H(p_\mathrm{D}) - H(p_\mathrm{I}) - H(p_\mathrm{S}) \ , \tag{5.12}$$

where $H(.)$ is the binary entropy. The Gallager model does not make any assumptions on the error patterns. Zigangirov later improved the Gallager bound [172], and recent research used simulation-based approaches to estimate the information rate [174]. But for FPSCC's deletion and insertion rates observed the differences are negligible and therefore we use the simpler Gallager bound.

Table 5.2: Average angle change rate (symbol rate) depending on the RTT

RTT (ms)	**Rate (symbols/s)**
25	10.3
75	6.3
125	4.8

From the experiments we examined the average number of deletions, insertions and substitutions, as if no synchronisation mechanism had been used. We measured deletion rates of 3–4% and insertion rates of 1–2% for bots and human players alike, independent of the RTT. In the following we assume average rates of $p_{\mathrm{D}} = 0.032$ and $p_{\mathrm{I}} = 0.016$.

The sole source of substitution errors were too large gaps between user commands (see Section 5.2.1), since the map has no teleporters or movers and pitch clamping did not occur. For fast clients that send user commands at least every 20 ms the substitution error rate would have been zero for our selected RTT values. However, there were very few gaps of 30 ms. The substitution error rate increases with increasing RTT, but even for 125 ms it is much smaller than the deletion and insertion rates ($p_{\mathrm{S}} = 0.0031$).

Based on Equation 5.12 we estimate the lower bound of the capacity in bits per symbol (angle change). The number of symbols depends on the RTT and the player behaviour. We computed the average symbol rate f_{S} for the different RTTs from the experimental data (see Table 5.2). With increasing RTT it takes longer for the players to kill each other and hence the rate does not proportionally decrease with increasing RTT.

Our model can be extended to include packet loss. Loss of snapshots causes bit deletions (snapshots lost for the receiver) and insertions (snapshots lost for the sender). The loss of snapshots for sender and receiver is independent and therefore both p_{D} and p_{I} are increased by the packet loss rate p_{L}.

Loss of user commands increases the chance of late user commands and hence increases p_{S}. Given the RTTs in our experiments bit errors are only caused if the last user command in a period between snapshots is lost; the loss of earlier commands does not matter. Therefore, p_{S} is increased by $\frac{1}{2}p_{\mathrm{L}}$ since even if a command is lost there is a 50% chance that the covert bits are correct (uniform random data).

In case of symmetric visibility without packet loss we model the channel as q-ary symmetric channel (q-SC). As before, N is the number of bits encoded per angle change. Then the number of symbols is $q = 2^N$. The capacity of the q-SC is [212]:

$$C = 1 - \frac{H(p_{\mathrm{S}}) - p_{\mathrm{S}} \log_2\left(2^N - 1\right)}{N} . \tag{5.13}$$

Note that for $N = 1$ the capacity of the q-SC is identical to the capacity of the BSC [22], and for $N = 2$ it is only marginally higher (< 0.03 bits/symbol given the relatively low p_{S}). Finally, if the capacity and symbol rate are known the maximum transmission

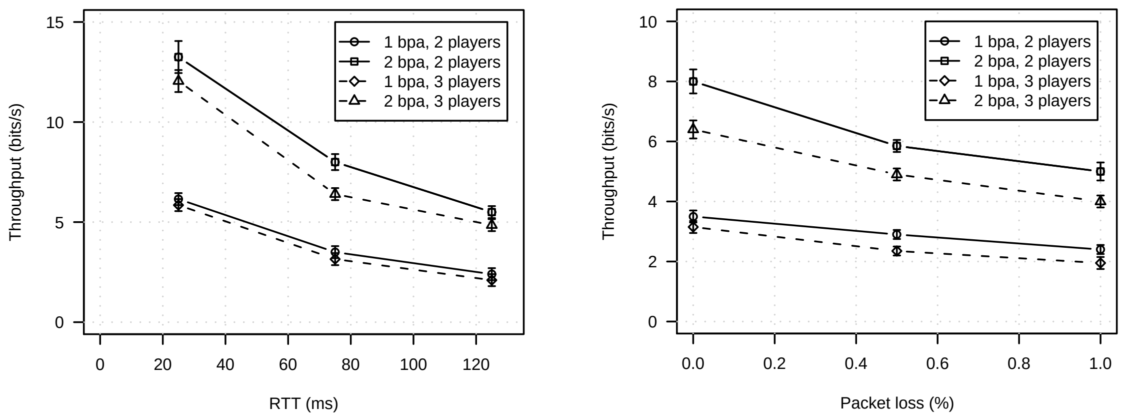


Figure 5.14: RFPSCC throughput depending on RTT (left) and packet loss rate (right)

rate of the channel in bits per second is:

$$R \geq N \cdot C \cdot f_S \,. \tag{5.14}$$

5.5.3 Bot players

We measured the throughput and bit error rate depending on the number of players, bits encoded per angle (bpa), RTT and packet loss rate. For each distinct parameter setting we let the bots play five one-hour games. In total we collected data for over 170 hours of game-play. The throughput in both directions (Alice to Bob, Bob to Alice) differs less than 0.1 bits/s and hence we only plot the means in the following graphs.

Figure 5.14(left) compares the average throughput over increasing RTT for 1 bpa and 2 bpa and two or three players with 0% packet loss. Figure 5.14(right) compares the average throughput over increasing loss rate for 1 bpa and 2 bpa and two or three players at an RTT of 75 ms. The error bars denote the standard deviation over the five games. The bit error rate was zero in all experiments.

The throughput of RFPSCC decreases relatively quickly with increasing RTT. However, it decreases less than expected. For example, the throughput for 75 ms should be 50% of the throughput for 25 ms, but the actual throughput reduces only to about 57% (from 6.15 bits/s to 3.5 bits/s). This is because the higher the latency, the longer the bots need to kill each other since aiming becomes more difficult. The same effect occurs for human players [202]. Enabling packet loss support causes a throughput reduction and all experiments with 0% loss had loss support disabled. Hence the throughput of RFPSCC decreases more rapidly between 0% and 0.5% and then the reduction is slower.

Figure 5.15 shows the throughput depending on packet loss for RTTs of 25 ms and 125 ms. The trends are similar to the results shown in Figure 5.14(right). Increasing RTT

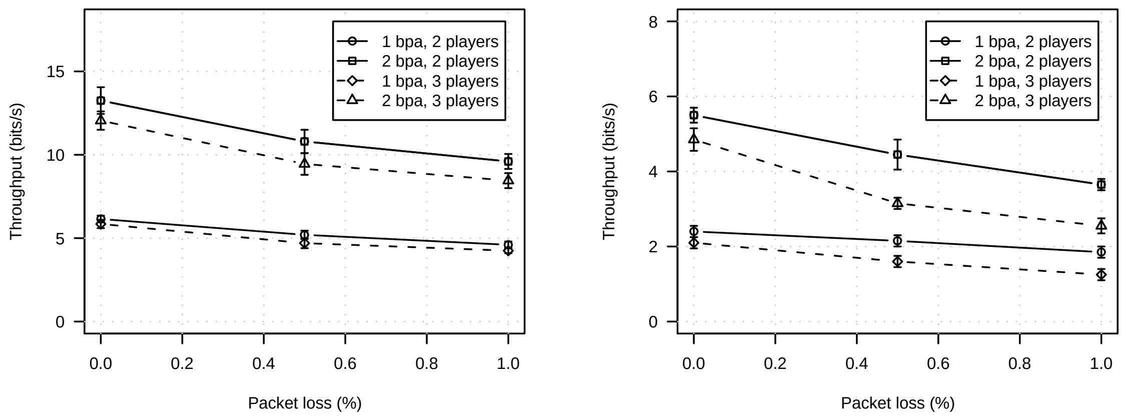


Figure 5.15: RFPSCC throughput depending on packet loss for an RTT of 25 ms (left) and 125 ms (right)

across a plausible range of values causes a more significant reduction in throughput than adding plausible levels of packet loss.

Because of limited resources we only tested RFPSCC with two and three bot players. The throughput reduces with increasing number of players and since the bots are unaware and unsupportive of RFPSCC this trend would continue for a larger number of players. However, if both Alice and Bob are players and covert sender/receiver, they could improve the throughput by staying in range of each other, which also is natural in recent team-based games if both are in the same team.

We also tested RFPSCC across the Internet. The two Q3/RFPSCC clients were at the same location as before. The Q3 server was located 15 hops away from the clients and the average RTT was 48–49 ms, as measured by traceroute and ping. The server was connected to the Internet via an ADSL2 link. The game traffic shared the ADSL2 link with other traffic, but the total amount of traffic was always much below the link capacity.

In some initial tests we measured snapshot loss rates between 0.0015% and 0.01% with only single snapshots lost each time (indicating our non-bursty testbed loss settings are justified). Hence we configured RFPSCC for a maximum loss burst of one snapshot. We performed three one-hour measurements with 1 bpa and three one-hour measurements with 2 bpa. The throughput was 3.15 ± 0.3 bits/s for 1 bpa and 6.25 ± 0.2 bits/s for 2 bpa (with zero bit errors). These results are consistent with the testbed measurements.

Table 5.3 compares the maximum transmission rates with the measured throughputs without packet loss for different delays (values rounded). RFPSCC achieves 77–90% (1 bpa) and 89–96% (2 bpa) of the capacity lower bound. The overhead is always approximately 0.6–0.7 bits/s. Leigh compared Gallager's capacity lower bound with the rate of efficient watermark codes, assuming zero substitution errors [173]. Given the error rates of FPSCC the best watermark code provides approximately 0.5 bits/symbol meaning it

Table 5.3: Comparison of the channel capacity with empirically measured throughputs without packet loss (bpa = bits per angle change)

RTT (ms)	Maximum rate (bits/s)		Throughput (bits/s)	
	1 bpa	2 bpa	1 bpa	2 bpa
25	6.9	13.8	6.2 (90%)	13.2 (96%)
75	4.2	8.4	3.5 (83%)	7.9 (94%)
125	3.1	6.2	2.4 (77%)	5.5 (89%)

Table 5.4: Comparison of the channel capacity with empirically measured throughputs with packet loss (bpa = bits per angle change)

RTT (ms)	Packet loss (%)	Maximum rate (bits/s)		Throughput (bits/s)	
		1 bpa	2 bpa	1 bpa	2 bpa
25	0.5	6.1	12.2	5.2 (85%)	10.8 (89%)
25	1	5.4	10.8	4.6 (85%)	9.6 (89%)
75	0.5	3.7	7.6	2.9 (78%)	5.9 (78%)
75	1	3.3	6.6	2.5 (76%)	5.1 (77%)
125	0.5	2.8	5.6	2.2 (77%)	4.4 (79%)
125	1	2.4	4.8	1.9 (77%)	3.7 (77%)

achieves approximately 75% of the capacity. This is less than what RFPSCC achieves for non-zero substitution error rates.

Table 5.4 compares the lower bound of the capacity with the actual throughput for different RTTs and packet loss rates. For low RTTs RFPSCC is quite efficient ($\geq$ 85%), but for higher RTTs the efficiency reduces to approximately 77%.

We conclude that RFPSCC is reliable. The throughput of RFPSCC is low, but it is of the same magnitude as the throughput of sophisticated packet timing covert channels providing 5–20 bits/s (see Chapter 4 and [104]). Since game sessions typically last from tens of minutes up to 1–2 hours [213], the overall amount of data that can be transmitted is substantial. For example, about 2.7 kbyte can be transmitted in one hour (assuming two players, 2 bpa encoding, an RTT of 75 ms and a packet loss rate of 0.5%).

5.5.4 Human players

We also performed five-minute games with nine different human players at an RTT of 25 ms without packet loss. All players used the same client for playing. Each time one human player played against one bot. Hence any variation of a player's behaviour is purely based on the player's actions and not on varying behaviour of the opponent. Figure 5.16 shows a boxplot comparing the throughput of bots (from the previous experiments) and human players.

Boxplots show the distributions of different variables using boxes with whiskers. The box shows the inter-quartile range (IQR) – the bottom end of a box denotes the first

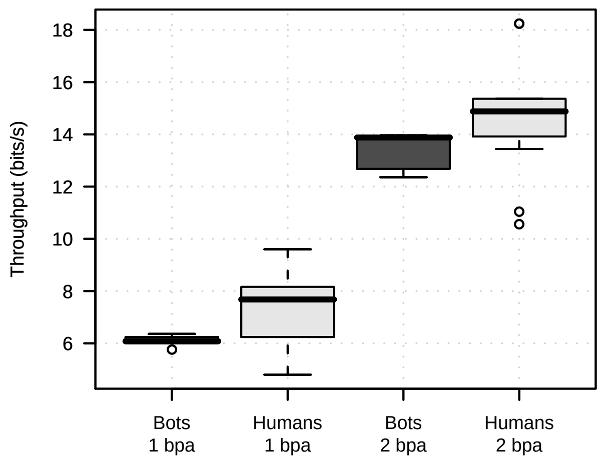


Figure 5.16: Comparison of throughput with bots and human players for 1 and 2 bits encoded per angle change

quartile, the bold line inside a box denotes the second quartile (median) and the top end of a box denotes the third quartile. Whiskers extend up to 1.5 times the IQR. Any values further away are deemed outliers and shown as little circles.

As expected, the variance of the throughput is much larger with human players as they have different playing styles. As predicted by the comparison of the angle changes per second for 1 bpa the throughput is up to 50% higher with human players, but on average the increase is only 20%. In general the throughput increases by 1–2 bits/s with human players. The bit error rate was zero in all experiments.

5.5.5 Symmetric visibility

We also measured the throughput for Q3 with symmetric visibility (see Section 5.3.1). In the absence of packet loss no synchronisation errors occur and hence we used simple and efficient block-size framing. But the channel is not error-free as substitution errors may occur, for example because of occasional large gaps between user commands. Therefore, we used a (48,40) RS code for error correction. A short code was selected in order to get data in small time intervals and we performed initial experiments to determine the amount of redundancy necessary to achieve a zero bit error rate. However, our choice remains somewhat ad-hoc, as we did not perform an extensive search for the best code.

Figure 5.17 compares the average throughput for symmetric visibility over increasing RTT for 1 bpa and 2 bpa and two or three players without packet loss. The error bars denote the standard deviation. Despite the overhead of the RS code the average throughput is 1–2 bits/s higher than the throughput for standard Q3 with RFPSCC (see Figure 5.14).

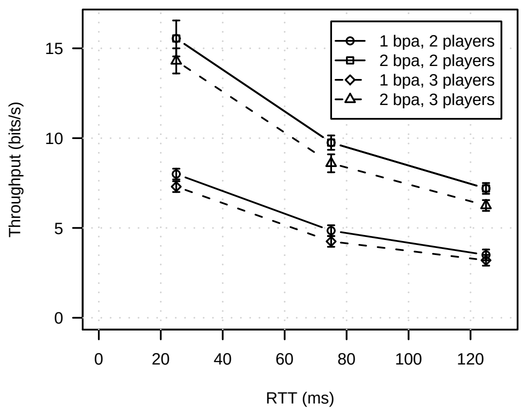


Figure 5.17: Throughput of FPSCC with (48,40) RS code depending on RTT if visibility between players is symmetric

Table 5.5: Comparison of the channel capacity with empirically measured throughputs for Q3 with symmetric visibility without packet loss (bpa = bits per angle change)

RTT (ms)	Maximum rate (bits/s)		Throughput (bits/s)	
	1 bpa	2 bpa	1 bpa	2 bpa
25	10.2	20.4	8.0 (78%)	15.6 (77%)
75	6.2	12.4	4.8 (77%)	9.8(79%)
125	4.7	9.4	3.6 (77%)	7.3(78%)

The results give an indication of the possible throughput for FPS games that have symmetric visibility. Note that the simple transport scheme for symmetric visibility cannot tolerate packet loss, as each lost snapshot could cause bit synchronisation errors. If packet loss can occur, RFPSCC must be used. However, other types of games may use TCP-based protocols and then for the covert channel there would be no lost packets.

Table 5.5 compares the maximum transmission rates with the measured throughputs for different delays (values rounded). FPSCC with symmetric visibility achieves 77–79% of the capacity. Given the overhead of the RS code and per-block sequence numbers FPSCC should achieve about 81% of the capacity. We think the difference of about 3% is caused by inefficiencies in the implementation.

We now discuss the main factors that contribute to the overhead of RFPSCC compared to the simple scheme for symmetric visibility. RFPSCC requires that both peers send SYNC symbols after visibility has been established. Even in the typical case where Alice and Bob synchronise as quickly as possible one snapshot is lost (up to $2N$ bits). In the worst case the overhead is much higher. At the end of frames both Alice and Bob roll back some bits that were sent. Up to $4N$ bits are lost because of the rollback and with packet loss support this increases to up to $(2+l_{\max})\cdot 2N$ bits.

Throughput is also reduced because with RFPSCC a dead peer cannot receive bits (see Section 5.3). Since the dead time is usually short we assume the peer still alive is usually visible for the whole time. Then the overhead depends on the average number of bits per second, the number of deaths per second and the time it takes for players to respawn (two seconds by default).

5.6 Conclusions

We developed and analysed a novel covert channel in first person shooter online game traffic (FPSCC). FPSCC encodes covert bits as imperceptible variations of a player's character's movements. FPSCC is not limited to FPS games, but could be applied to any game where player movements of one or more in-game characters are regularly propagated to other players and their game clients. We empirically evaluated FPSCC using a proof-of-concept implementation based on Quake III Arena.

One major problem to overcome was bit synchronisation errors on the channel caused by asymmetric state exchange and packet loss. We developed a mechanism for reliable transport tailored to FPSCC (called RFPSCC) that eliminates bit synchronisation errors and has low overhead.

We analysed the throughput and bit error rate of RFPSCC in a testbed with different network delays and packet loss rates as well as across the Internet. We also varied the number of bits encoded per angle change and the number of players on the map. We found the channel provides throughputs of up to over 15 bits/s with human players. The bit error rate is zero, even with modest packet loss on the overt channel.

We also developed a variant of FPSCC for symmetric state exchange requiring no synchronisation mechanism that works in the absence of packet loss. We evaluated this variant with a modified game server and measured throughputs of 1–2 bits/s higher than for RFPSCC with zero bit error rate. We developed an information-theoretic model to estimate the channel's capacity. We found the throughput measured for RFPSCC is always over 77% and up to over 90% of the capacity, suggesting that RFPSCC is quite efficient.

RFPSCC's throughput is lower than that of the network-layer channels analysed in the previous chapters. However, it is sufficient to covertly engage in low-speed text messaging or general data transfers. Key advantages of RFPSCC for users are that it is an indirect channel and it cannot be eliminated without eliminating the overt channel (the game). However, as we will show in Chapter 7 the channel can still be detected.

5.6.1 Future work

While our current results are very encouraging, more trials with more players, different maps and game settings could be done to enhance our understanding of (R)FPSCC's limitations and performance. Most of our work focused on FPSCC in unicast mode, the use of FPSCC in broadcast mode is left for further study. Furthermore, the application of similar channels to non-FPS online multiplayer games or immersive virtual worlds remains future research.

RFPSCC is quite efficient for small RTTs as our comparison with the channel capacity shows, but for larger RTTs the performance decreases significantly. An improved encoding technique should be developed to increase the performance for larger RTTs. Another avenue of new research is to make the channel stealthier. While it appears to be difficult to make the channel hard to detect for Quake III Arena, for other games or immersive worlds it may be less challenging.

FPSCC is a channel between two game clients. There could be a variant of FPSCC utilising only the user commands and snapshots exchanged directly between a client and the server. Usually FPS game servers are dedicated servers, but players can simultaneously run a client and a server on their computer. We leave an implementation and analysis of this channel for future work.

Chapter 6
Temperature-based Covert Channels

In this chapter we investigate noisy indirect timing channels that transmit covert data via changes of temperature (referred to as temperature-based covert channels), exploiting the fact that a host's CPU temperature depends on the number of service requests processed per time unit and the skew of a host's system clock depends on the temperature. Clock skew changes can be estimated remotely from a series of a clock's timestamps.

The initial purpose of such channels was to reveal hidden services [44] – services (e.g. web servers) hidden inside anonymisation networks (e.g. the Tor network [214]) providing mutual anonymity[1]. The attacker identifies the hidden service by correlating variable clock-skew patterns measured over time for several candidates with the request-rate pattern sent to the hidden service over the anonymisation network.

Temperature-based channels could possibly be used for general-purpose covert communications, even in scenarios where most other simpler covert channels are not available, because it is difficult to eliminate them completely [44]. They are not trivial to detect and their indirect nature makes it harder for an adversary to discover the link between covert sender and receiver.

We first provide some background on Tor hidden services and outline previous attacks against them. Then we describe how temperature-based covert channels work. Next, we develop a novel technique to minimise one key source of channel noise inherent in the clock-skew estimation. This technique not only makes the channel much more effective because of an increased capacity. It also enables new more efficient attacks against Tor hidden services, which we then describe.

Our new attacks are based on the correlation of clock skew measured for the hidden service across Tor and clock skew measured directly for several candidates. High-frequency timestamps are not available across Tor and only our improved channel enables accurate clock-skew estimation from low-frequency timestamps, such as HTTP timestamps. The amount of traffic exchanged is reduced significantly, since remote clock-skew estimation generates much less traffic than remote modulation of CPU load.

We evaluate the effectiveness of the improved channel and the new attacks in a testbed as well as over the Internet. The results show that the channel's noise is significantly

[1]The identity of the service operator is not revealed to the user and the identity of the user is not revealed to the service operator.

reduced by up to two orders of magnitude. We also demonstrate that our new attacks works; it identifies the hidden server in less than 2.5 hours. We focus on Tor, but our results should be applicable to other low-latency hidden service designs.

Finally, we propose a method to estimate the capacity of the channel. Knowing the capacity is very important for determining whether the channel poses a threat in certain scenarios and actions should be taken against it. We compute upper bounds of the capacity based on empirical data collected for example intermediate hosts. The results show that the capacity is limited to 10–20 bits per hour.

The low capacity makes the channel less relevant for general-purpose communication. However, the capacity is more than sufficient for attacking hidden servers where only a few bits need to be transmitted. Murdoch's attack could be executed with reasonably probability for false positives in only 1–2 hours, much quicker than assumed in [44]. In this case the channel requires handling (elimination, capacity limitation or monitoring).

6.1 Background

First we describe previous attacks against Tor hidden services. Then we explain how temperature-based covert channels work. Next, we discuss a key component of the channel – the remote estimation of clock skew. Finally, we provide an overview of remotely accessible clocks, required for clock-skew estimation.

6.1.1 Existing attacks against Tor

The Tor network [214] is the latest generation of the Onion Router Project [215]. Tor is a popular, deployed system and as of January 2008 there are about 2 500 active Tor servers. The Tor hidden service facility allows the provision of pseudonymous services, protecting the owners' identity and also resisting selective DoS attacks [214].

High-profile examples where this feature would have been valuable include blogs whose authors are at risk of legal attacks [216, 217]. Other hidden web sites host suppressed documents, permit the submission of leaked material, and distribute software written under a pseudonym.

Since Tor is an overlay network, servers hosting hidden services are accessible both directly and over the anonymous channel. Traffic patterns through one channel have observable effects on the other, thus allowing a service's pseudonymous identity and IP address to be linked.

Øverlier and Syverson showed that a hidden service could be located rapidly because a Tor hidden server selects nodes at random to build connections [218]. The attacker repeatedly connects to the hidden service, and eventually a node she controls will be the one closest to the hidden server. By correlating input and output traffic, the attacker can discover the server's IP address.

Murdoch and Danezis presented an attack where the target visits an attacker-controlled web site, which induces traffic patterns on the Tor circuit protecting the client [219]. Simultaneously, the attacker probes the latency of all the publicly listed Tor nodes and looks for correlations between the induced pattern and observed latencies. When there is a match, the attacker knows that the node is on the target circuit, and so she can reconstruct the path, although not discover the end node.

Murdoch proposed the most recent attack based on temperature-based covert channels [44]. This attack is described below.

6.1.2 Temperature-based covert channel

In a temperature-based covert channel Alice modulates the CPU load of an unwitting intermediate host connected to a network (e.g. a public server) by varying the rate of requests sent to it based on the covert bits to be encoded. The varying request rate will cause CPU load changes on the intermediate host. The change in CPU load changes the temperature, which in turn induces changes in clock skew – the deviation of the intermediate's clock from the true time. Bob measures the intermediate's clock skew by obtaining timestamps from the intermediate's clock and comparing these against a local clock. Bob then decodes the covert bits by estimating the clock-skew changes [44].

Two scenarios are possible. In the first scenario both Alice and Bob are separated from the intermediate by a network. Alice and Bob could be controlled by the same person (e.g. attacking Tor hidden services), or could be different persons (e.g. general covert communications). In the second scenario Alice is located on the intermediate host itself and manipulates the CPU load directly; only Bob is separated by a network. This is a possible scenario for the ex-filtration of sensitive information. The scenarios are depicted in Figures 6.1 and 6.2.

When attacking hidden services the *attacker* takes the role of Alice and Bob, and the intermediate host is the *target* [44]. The attacker induces a load pattern on the target by frequently accessing the hidden service via the anonymisation network or staying silent, effectively sending a pseudo-random bit sequence. At the same time the attacker measures the clock skew of a set of candidate hosts. The attacker has identified the hidden server if the same bit sequence is recovered from the clock-skew measurement of a candidate.

6.1.3 Clock-skew estimation

All networked devices have clocks constructed from hardware and software components. A clock consists of a crystal oscillator that ticks at a nominal frequency and a counter

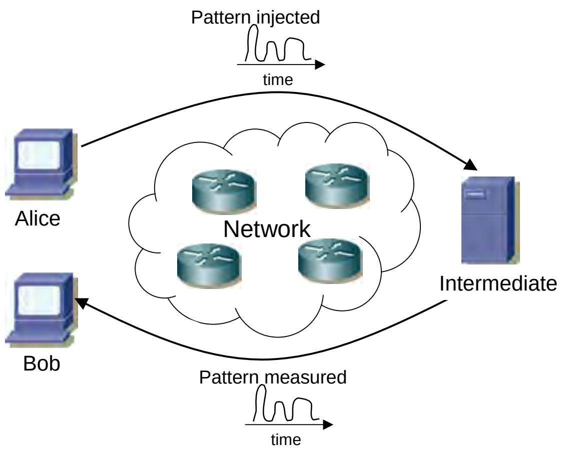


Figure 6.1: Temperature-based covert channel where Alice and Bob are separated by a network

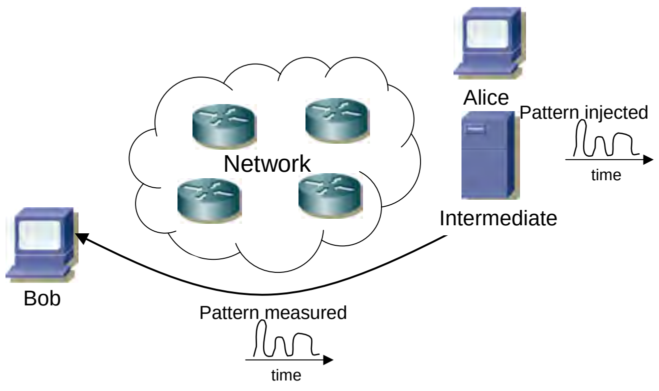


Figure 6.2: Temperature-based covert channel where Alice induces load directly on the intermediate host

that counts the number of ticks. The actual frequency of a device's clock depends on the environment, such as the temperature and humidity, as well as the type of crystal.

It is not possible to directly measure a remote host's true clock skew. However, the attacker can measure the offset between the target's clock and a local clock, and then estimate the relative clock skew. For a packet *i,* containing a timestamp of the target's clock received by the attacker, the offset $\tilde{o}_i$ is [44]:

$$\tilde{o}_i = \tilde{t}_i - t_{\mathrm{r}_i} = s_{\mathrm{c}} t_{\mathrm{r}_i} + \int_0^{t_{r_i}} s(t)dt - c_i/h - d_i \,, \tag{6.1}$$

where $\tilde{t}_i$ is the estimated time at the intermediate, t_{r_i} is the local time the packet was received, s_{c} is the constant clock-skew component, the integral over $s(t)$ is the variable clock skew component, c_i/h is the quantisation noise for random sampling and d_i is the network delay.

The constant clock skew is estimated by fitting a line above all points $\tilde{o}_i$ while minimising the distance between each point and the line above it using the linear program-

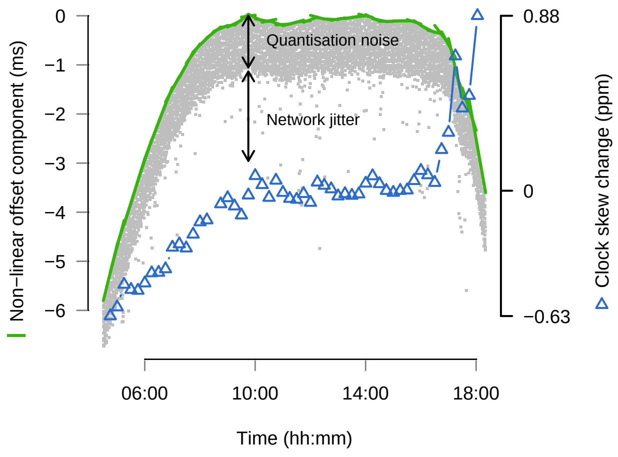


Figure 6.3: Estimating the variable clock skew

ming algorithm described in [220]. This leaves the variable part of the clock skew and the noise. To estimate the variable clock skew per time interval, the same linear programming algorithm is used for each time window of size w.

Figure 6.3 shows an example of a clock skew measurement across the Internet. The target was 22 hops away with an average RTT of 325 ms. The measurements were taken from the TCP timestamp clock ticking with a frequency of 1 kHz (see Section 6.1.4)[2]. The constant clock skew s_c has already been removed. The dots ($\cdot$) are the offsets between the two clocks, the line ($-$) on top is the piece-wise estimation of the variable skew and the triangles ($\triangle$) are the negated values of the derivative of the variable clock skew. To be consistent with [44] in this chapter we always plot the *negated* variable clock skew.

The noise apparent in the figure has two components: the network jitter on the path from the attacker to the target and the timestamp quantisation error. Note that the network jitter also contains noise inherent in the attacker's measurement of when packets are received, and noise caused by variable delay at the intermediate host between generating the timestamps and sending the packets. In Figure 6.3, we clearly see the 1 ms quantisation noise band below the estimated slope caused by the 1 kHz clock. Offsets below this band were also affected by network jitter.

The samples close to the slope on top are the samples obtained immediately after clock ticks with negligible network jitter. The samples at the bottom of the quantisation noise band are samples obtained immediately before clock ticks. With the linear programming algorithm, only the samples close to the slope on top contribute to the accuracy of the measurement. Assuming an uncongested network, the network jitter is skewed towards zero and often small even on long-distance links (see Section 6.4.2). The quantisation noise is inversely proportional to the frequency of the intermediate's clock. Depending

[2]No additional CPU load was generated on the intermediate host during the measurement.

on the timestamps used and the intermediate's operating system, the clock frequency is typically between 1 Hz and 1 kHz, resulting in a noise band between 1 s and 1 ms. If the target does not expose a high-frequency clock, the quantisation noise can be significantly larger than the noise caused by network jitter.

To increase the accuracy of the measurement in the presence of high quantisation noise, w must be set to larger values, as the probability of getting samples close to the slope on top increases with the number of samples. However, large windows only allow very infrequent measurements. Oversampling provides more frequent results while keeping w large to minimise the error. Without oversampling the time windows do not overlap and the start times of the windows are $S = \{0, w, 2w, \ldots, nw\}$. With oversampling, the windows overlap and the windows start at times $S = \{0, w/o, 2w/o, \ldots, nw/o\}$, where o is the oversample factor.

However, over-sampling with large w still has drawbacks. The first estimate is obtained only after $w/2$ regardless of o, since clock-skew estimates are always assigned to the middle of the time windows. This means for large w it is impossible to get estimates close to the start and end of measurements. Furthermore, averaging caused by large windows makes it impossible to measure steep clock-skew changes accurately, for example when a CPU load inducement starts and temperature increases quickly [44].

6.1.4 Timestamp sources

Previous research used different timestamps for clock-skew estimation: ICMP timestamps, TCP timestamps or TCP sequence numbers [221, 44].

TCP sequence numbers on Linux are the sum of a cryptographic result and a 1 MHz clock. They provide good clock-skew estimates over short periods because of the high frequency, but re-keying of the cryptographic function every five minutes makes longer measurements more difficult [44].

ICMP timestamps have a fixed frequency of 1 kHz. Their disadvantage is that they are affected by clock adjustments done by the Network Time Protocol (NTP) [222], which makes estimation of variable clock skew more difficult. Furthermore, ICMP messages are now blocked by many firewalls.

TCP timestamps have a frequency between 1 Hz and 1 kHz, depending on the operating system. They are currently the best option for clock-skew measurements because they are widely available and unaffected by NTP adjustments on Linux, FreeBSD and Windows [221]. However, even TCP timestamps are not always available. They may not be enabled on certain operating systems and they cannot be used if there is no end-to-end TCP connection to the intermediate host, i.e. across the Tor network.

HTTP timestamps have a frequency of 1 Hz and are available from every web server. However, they have not been used previously for clock-skew measurements due to the low frequency. We describe how to exploit them in Section 6.3.

6.2 Improved covert channel

In previous research, the clock skew was remotely measured by random sampling of the clock [221, 44]. To minimise the quantisation error, Murdoch proposed to use synchronised sampling instead of random sampling [44]. Here, the attacker synchronises the timestamp requests with the target's clock ticks, attempting to obtain timestamps immediately after clock ticks, where the quantisation error is smallest. Synchronised sampling improves the accuracy, especially for low-resolution timestamps.

We first describe how synchronised sampling improves the efficiency of the existing covert channel. Next, we describe the novel synchronised-sampling technique in detail. Finally, we discuss possible errors that reduce its accuracy.

6.2.1 Synchronised sampling

To prevent high quantisation noise the attacker must probe the target's clock immediately after clock ticks occurred, because here the quantisation error is smallest. To achieve this, the attacker has to synchronise its probing frequency and phase with the remote clock. We assume the attacker selects a nominal sample frequency, based on the desired accuracy and intrusiveness of the measurement.

The attacker cannot measure the exact time difference between the arrival of probe packets and the clock ticks. To maintain synchronisation, the attacker has to alternate between sampling the clock before and after a clock tick. Only samples taken after the clock tick are used for the clock-skew estimation.

Samples taken before the clock tick can be corrected by adding one tick, as their true value is actually closer to the next clock tick. However, the linear programming algorithm still cannot use these samples because for them the quantisation error and network jitter are in opposite directions and cannot be separated.

Figure 6.4 illustrates the benefit of synchronised sampling. The solid step line is the target's clock value over time and the dashed line is the true time. Random samples are distributed uniformly between clock ticks whereas synchronised samples are taken close to them[3]. The quantisation errors are the differences between samples' y-values and the true time. The absolute quantisation errors are shown as bars at the bottom. Synchronised sampling leads to smaller errors in comparison with random sampling.

[3]Times for samples before the tick have been corrected as described earlier.

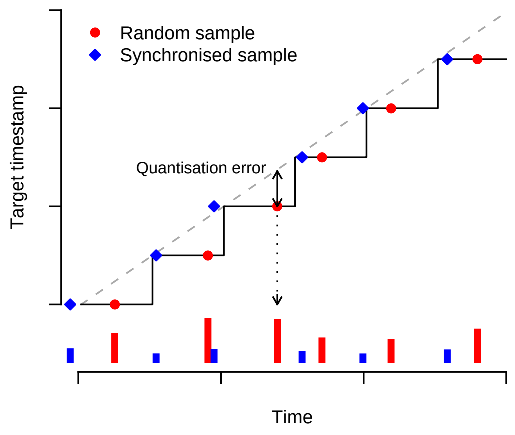


Figure 6.4: Advantage of synchronised sampling over random sampling

Our algorithm is similar to existing Phase Lock Loop (PLL) techniques used for aligning the frequency and phase of a signal with a reference [223]. However, whereas PLL techniques measure the phase difference of two signals, we can only estimate the phase difference by detecting whether a sample was taken before or after a clock tick.

6.2.2 Algorithm

Initially, the attacker starts probing with the nominal sampling frequency and measures how many clock ticks occur in one sample interval (`target_ticks_per_interval`). The measurement is repeated to obtain the correct number of ticks.

The attacker cannot measure the exact time difference between the arrival of a probe and the target's clock tick. However, the attacker can measure the position of the probe's arrival relative to the target's clock tick based on the number of clock ticks that occurred between the current and the last timestamp (`ticks_diff`). If the number of clock ticks is less than `target_ticks_per_interval`, the sample was taken before the tick and vice versa. If `ticks_diff` equals `target_ticks_per_interval` the position is unchanged. At the start of the measurement the position is not known and the attacker needs to continuously increase or decrease the probe interval until a change occurs.

The probe interval, the reciprocal of the probe frequency, is controlled using the following mechanism (see Algorithm 6.1). It is adjusted based on the position error each time a position change occurs and the previous position was known using a Proportional Integral Derivative (PID) controller [224]. PID controllers base the adjustment not only on the last error value, but also on the magnitude and the duration of the error (integral part) as well as the slope of the error over time (derivative part). K_p, K_i and K_d are predefined constants of the PID controller, dimensioned based on initial experiments.

Alternatively, the linear programming algorithm [220] is used to estimate the relative clock skew between attacker and target based on a sliding window of timestamps, and the probe interval is adjusted based on the relative clock skew. This technique works well if the estimates are accurate (e.g. high-frequency clocks and low network jitter).

Algorithm 6.1 Probe interval control

```
function probe_interval_adjustment(pos, last_pos)
  if pos $\neq$ last_pos and pos $\neq$ UNKNOWN and
     last_pos $\neq$ UNKNOWN then
    return $K_p$·(last_adj_before + last_adj_behind) +
           $K_i$·(integ_adj_before + integ_adj_behind) +
           $K_d$·(deriv_adj_before + deriv_adj_behind)
  else
    return 0
```

In order to maintain synchronisation, the attacker has to enforce regular position changes. This is done by modifying the time the next probe will be sent. If the current position is before the clock tick the send time of the next probe is increased based on the last adjustment `last_adj_before`. If the current position is behind the clock tick the next probe's send time is decreased based on the last adjustment `last_adj_behind`. The adjustments are modified based on how well the attacker is synchronised.

If a change of position occurs between two samples, the difference between the arrival of the probe packet and the target's clock tick is smaller than the last adjustment and therefore the next adjustment is decreased. If no position change occurs the error is assumed to be larger than the last adjustment and the next adjustment is increased. The initial adjustment is a pre-defined constant. Algorithm 6.2 shows the probe send time adjustment algorithm. Parameters α and β are pre-defined constants that determine how quickly the algorithm reacts ($0 < \alpha < 1$ and $\beta > 1$).

Algorithm 6.2 Next probe send time adjustment

```
function next_probe_time_adjustment(pos, last_pos)
  if pos = BEFORE then
    last_adj = last_adj_before
  else
    last_adj = last_adj_behind

  if pos $\neq$ last_pos then
    return $\alpha$·last_adj
  else
    return $\beta$·last_adj
```

The probe interval and send-time adjustments are limited to a range between pre-defined minimum and maximum values to avoid very small or very large changes. If packet loss is detected, the algorithm adjusts `ticks_diff` by subtracting the number of lost packets multiplied by `target_ticks_per_interval`. Reordered packets are considered lost. Loss of responses is detected using sequence numbers. A sequence number is embedded into each probe packet such that the target will return the sequence number in the corresponding response.

The actual sequence number encoding depends on the protocol used for probing. For ICMP the sequence number is the ICMP Identification field whereas for TCP the sequence number is the TCP sequence number field. For HTTP it is not possible to embed a sequence number into the protocol directly. Instead, sequence numbers are realised by making requests cycling through a set of URLs. A sequence number is associated with each URL. HTTP responses are mapped to HTTP requests using the known content length for each object[4].

Algorithm 6.3 shows the synchronisation procedure. Our algorithm works with different timestamps and different clock frequencies. It has been tested with ICMP, TCP and HTTP timestamps and TCP clock frequencies of 100 Hz, 250 Hz and 1 kHz.

Algorithm 6.3 Synchronised sampling

```
foreach response_packet do
  diff = target_timestamp − last_target_timestamp

  if target_ticks_per_interval = NA then
    pos = UNKOWN
    target_ticks_per_interval = ticks_diff
  else if ticks_diff > target_ticks_per_interval then
    pos = BEHIND
  else if ticks_diff < target_ticks_per_interval then
    pos = BEFORE
  else
    pos = last_pos

  probe_interval = probe_interval +
            probe_interval_adjustment(pos, last_pos)
  probe_time = last_probe_time + probe_interval +
          next_probe_time_adjustment(pos, last_pos)

  last_pos = pos
  last_target_timestamp = target_timestamp
  last_probe_time = probe_time
```

6.2.3 Errors

Constant delay on the path from the attacker to the target has no effect, but changes in delay affect the synchronisation process. In Appendix E.1 we provide more details on the sources of variable delays that affect the synchronisation process. They include jitter in

[4]We assume there are multiple objects with different content lengths accessible on the web server.

the sending process (jitter at the attacker), network jitter (queuing delays in routers) or jitter in the target's packet receiving process.

Often we can assume the network is uncongested and therefore network jitter is skewed towards zero. This is usually the case in a LAN (see Section 6.4). Even on long paths across the Internet jitter is often small, as many links are not heavily utilised and path changes are not frequent. However, when measuring clock skew over a Tor circuit we expect much higher network jitter. A Tor circuit is composed of a number of network connections between different nodes. The overall jitter does not only include the jitter of each connection but also the jitter introduced by the Tor nodes themselves.

The timing of probes is not very exact if the attacker is a userspace application. Even if the `send()` system call is executed at the appropriate time, there is a delay before the packet is actually sent onto the physical medium. The variable part of this delay affects the synchronisation process. The error could be reduced by running the software as kernel module, using a real-time operating system or using special network cards capable of very precise sending of packets.

Any variable delay in the packet receiving process of the target has the same effect and is unfortunately out of control of the attacker. The only way an attacker could reduce such errors would be to adjust the sending of the probe packets based on a prediction of the jitter inside the target, which appears to be a challenging task.

Another error is introduced at the attacker when timestamping the probe response packets. Furthermore, an error is introduced when the relative clock skew between attacker and target changes due to NTP and the algorithms needs to adjust the probe frequency. However, the attacker can control its time keeping and avoid sudden clock changes, and as described earlier some timestamps are unaffected by NTP.

6.3 New attacks

We first define the threat model. Then we describe the new attacks against Tor. Finally, we describe how HTTP timestamps are used in the new attacks.

6.3.1 Threat model

We assume that the attacker's goal is to link the hidden service pseudonym to the identity of its operator, which in practice can be derived from the server's IP address. Our attacker cannot observe, inject, delete or modify any network traffic, other than that to or from her own computer. The attacks we present here do not require control of any Tor node. However, we do assume that our attacker can access hidden services, which means she is running a client connected to a Tor network.

We also assume that our attacker has a reasonably limited number of candidate hosts for the hidden service (say, a few hundred). To mask traffic associated with hidden services, many of their hosts are also publicly advertised Tor nodes, so this scenario is plausible. All of our attack scenarios, with one notable exception, require that the attacker can access the candidate hosts directly via their IP address. Again, since many hidden servers are also Tor nodes, it is plausible that at least the Tor application is accessible.

6.3.2 New attacks against Tor

A major disadvantage of the attack in [44] is that the attacker needs to exchange large amounts of traffic with the hidden service across the Tor network to induce CPU load changes. It may not be possible to actually send sufficient traffic because Tor does not provide enough bandwidth, or because the service operator actively limits the request rate to avoid overload, prevent DoS attacks etc. Furthermore, the attack also relies on an exposed high-frequency clock on the target for adequate clock-skew estimation.

Synchronised sampling improves the existing attack by reducing the duration and hence the amount of network traffic required. It also makes the existing attack applicable in situations where high-resolution timestamps are not available. Furthermore, it opens the door for new variants of the attack based on clock-skew measurements across the Tor network.

High-frequency ICMP or TCP timestamps are not available across Tor, since it only supports TCP and streams are re-assembled on the client, removing any headers. But synchronised sampling can use low-frequency HTTP timestamps obtained from a hidden web server. The new attacks do not require generating load on the target and hence the amount of traffic exchanged is greatly reduced, as measuring clock-skew requires only a small amount of traffic; the exchange of one request/response every 1.5 seconds is sufficient.

In the first new attack variant, the attacker measures the variable clock skew of the hidden service via Tor, and of all the candidate hosts directly[5]. Then the attacker compares the variable clock-skew pattern of the hidden service with the patterns of all candidates. The variable clock skew patterns of different hosts differ sufficiently over time (see Section 6.4.3), and the duration of the attack could be increased arbitrarily.

A quicker version of this attack could only compare the fixed clock-skew of the target measured via Tor with the fixed clock-skew measured directly for all candidates. Kohno *et al.* showed that the fixed clock skew of a particular host changes very little over time, but the difference between different hosts is significant [221].

Another new attack variant is based on the idea of using clock-skew estimates for geo-location [44]. The attacker identifies the location of the candidates based on their IP

[5]Both timestamp sources must be derived from the same clock.

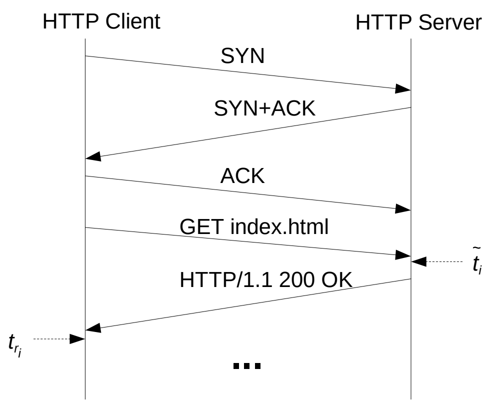


Figure 6.5: HTTP request/response and the timestamps used for clock-skew estimation

addresses and a geo-location database, such as GeoLite [225]. The attacker also measures the variable clock skew of the hidden service via Tor and estimates the location based on the variable clock-skew pattern. For example, the longitude can be estimated based on temperature peaks and troughs, and the latitude can be estimated by estimating the day length [44]. This attack works even when the attacker cannot access timestamps of the candidate hosts directly. However, it does not allow an unambiguous identification if candidate locations are geographically close together.

6.3.3 HTTP timestamps

The attacker acts as HTTP client sending minimal HTTP requests to the target. All recent HTTP servers include a 1 Hz timestamp in the Date header of HTTP responses because it is mandatory for HTTP 1.1 [226]. The HTTP timestamp is usually generated after the server has received the client's request[6]. The corresponding client timestamp is the time the packet containing the Date header is received, which is usually the first packet sent by the server after the TCP connection has been established (see Figure 6.5).

The client should open a TCP connection well in advance of needing to send the HTTP request to avoid the HTTP request being delayed by TCP's three-way connection establishment handshake, and ideally keep the connection open for the entire measurement period. However, HTTP servers may close the TCP connection after sending a reply. Therefore, the client should immediately re-open a torn-down TCP connection, enabling the next HTTP request to be sent at the right time, as determined by the synchronised sampling algorithm.

[6]We verified this for Apache 2.2.4 [227].

6.4 Empirical evaluation

First we compare the accuracy of synchronised and random sampling in a LAN testbed using TCP timestamps with typical clock frequencies of 100 Hz, 250 Hz and 1000 Hz, as well as 1 Hz HTTP timestamps. Since within the LAN network jitter is negligible the results show the maximum improvement of using synchronised sampling. Next, we compare the accuracy of synchronised and random sampling based on TCP timestamps across a 22-hop Internet path.

Then we compare the accuracy of synchronised and random sampling when probing a web server running as a Tor hidden service. We also show that a hidden web server can be identified among a candidate set by comparing the variable clock skew over time using synchronised sampling. Furthermore, synchronised sampling shows daily temperature patterns usable for geo-location that could not be identified using random sampling.

Finally, we investigate how long our synchronised probing technique needs for the initial synchronisation, which is the time until the attacker has locked on to the phase and frequency of the target's clock ticks. We compare the times for probing a web server in a LAN and probing a hidden web server over a Tor network.

It is impossible to measure the true values of the variable clock skew. We evaluate the accuracy of synchronised and random sampling against a baseline, estimated from timestamps with a precision of 1 μs, for which the quantisation error is negligible. In our tests a UDP client on the attacker sends packets to a UDP server running on the target. For each 'request' received the UDP server returns a packet with a timestamp set to the send time of the 'response'. The UDP client records the time it receives a response. We estimate the variable clock skew for this timestamp series using synchronised sampling. We refer to this as UDP probing or UDP measurement.

We compare the variable skew estimates for synchronised and random sampling with the reference values from the UDP measurement, using the root mean square error (RMSE) of the data values x against the reference values $\hat{x}$:

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_i (\hat{x}_i - x_i)^2}\,. \tag{6.2}$$

We also compute histograms of the noise band for synchronised and random sampling. The noise is defined as difference between the variable clock offsets and the variable clock-skew estimated from the UDP measurement. For random sampling the quantisation noise band is uniform with width $1/f$, where f is the clock frequency. For synchronised sampling it depends on how well the algorithm tracks the target's clock tick.

In all experiments we set $\alpha = 0.5$ and $\beta = 1.5$. For TCP timestamps the linear programming algorithm was used to adjust the probe interval with a sliding window of size 120

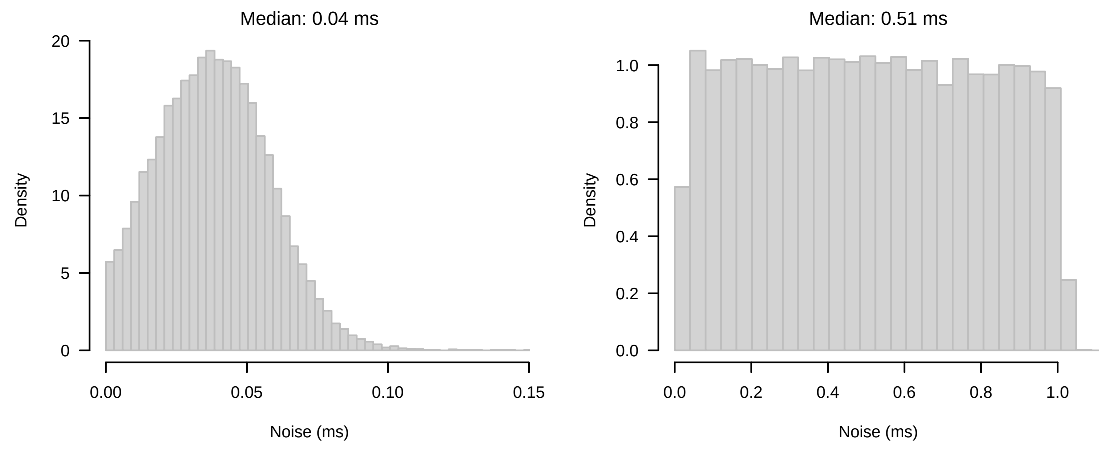


Figure 6.6: Noise distributions in LAN: synchronised sampling (left) and random sampling (right)

(LAN) and 300 (Internet). For HTTP timestamp measurements the probe interval was adjusted using a PID controller with $K_\mathrm{p} = 0.09$, $K_\mathrm{i} = 0.0026$ and $K_\mathrm{d} = 0.02$. The oversampling factor was chosen such that the time between two clock-skew estimates was always 30 s regardless of the window size. This has the advantage of providing approximately the same number of estimates for different window sizes[7].

6.4.1 Synchronised and random sampling within LAN

The attacker and target with a TCP clock frequency of 1 kHz were connected to the same Gigabit Ethernet switch[8]. The attacker simultaneously performed synchronised, random and UDP probing. Synchronised and random probing had an average sampling period of 1.5 s, the same period as in [44]. The UDP probing was performed with a faster average sample rate of 1 Hz to achieve a higher accuracy for the reference measurement. A second UDP measurement was carried out in order to investigate the error between UDP measurements. The experiment lasted approximately 24 hours.

Inside the LAN the average RTT was only 130 µs and the RTT/2 jitter was small with a maximum of 60 µs and a median of 30 µs. Figure 6.6 shows histograms of the noise bands of synchronised and random sampling. For synchronised sampling most of the offsets are within 100 µs whereas for random sampling there is the expected 1 ms noise band.

Figure 6.7 compares the RMSEs of synchronised sampling, random sampling and the second UDP measurement for different window sizes against the UDP reference with maximum window size (1800 s). It also compares the UDP reference against itself at smaller window sizes.

[7]For smaller windows there are still more samples at the start and end of measurement periods.
[8]Attacker and target were PCs with Intel Xeon 3.6 GHz Quad-Core CPUs, both running Linux 2.6.

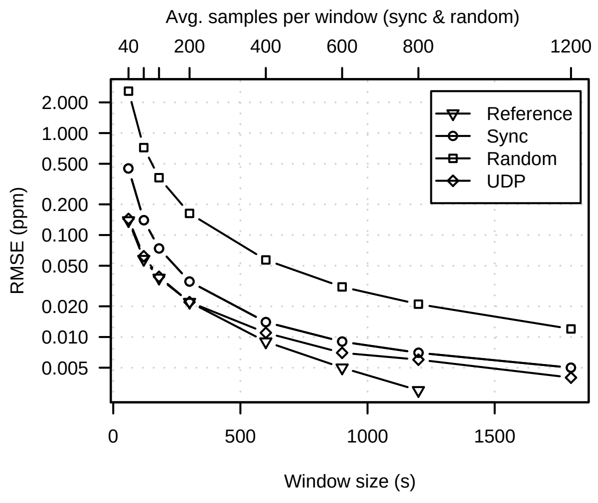


Figure 6.7: RMSEs of synchronised, random sampling and UDP reference in LAN with a target clock frequency of 1 kHz (log *y*-axis)

The results show that synchronised sampling performs significantly better than random sampling. There is a difference between the second UDP measurement and the UDP reference, but it is smaller than the difference between synchronised sampling and the UDP reference. The error of a UDP measurement is sufficiently small for using it as baseline, and in further experiments we carried out only one UDP measurement.

The target's clock frequency was 1 kHz, which is the maximum TCP clock frequency of current operating systems. However, in reality many hosts actually have lower TCP clock frequencies. For example, 100 Hz is the frequency used by older Linux and FreeBSD kernels and 250 Hz is the frequency of recent Linux 2.6 kernels.

We used the same setup to evaluate the RMSEs for lower clock frequencies. This time we ran three synchronised and three random probing processes simultaneously for 24 hours, rounding the target's timestamps so that we effectively measured 100 Hz, 250 Hz and 1 kHz clocks. Figure 6.8 shows the RMSEs for synchronised and random sampling for the different clock frequencies. The UDP measurement is omitted for better readability. The results show that the RMSE for random sampling increases significantly for lower clock frequencies, but the accuracy of synchronised sampling does not depend on the clock frequency.

In another LAN experiment we ran an Apache 2.2.4 web server on the target and the attacker used HTTP probing. The average probing interval was 2 s. The web server was completely idle, except for the requests generated by the attacker. The duration of the experiment was approximately 24 hours.

Although the experiment was carried out between the same two hosts as before, the RTT/2 jitter was higher with a maximum of 120 µs and a median of 60 µs. The web server running in userspace introduced the additional jitter. Figure 6.9 shows the noise for

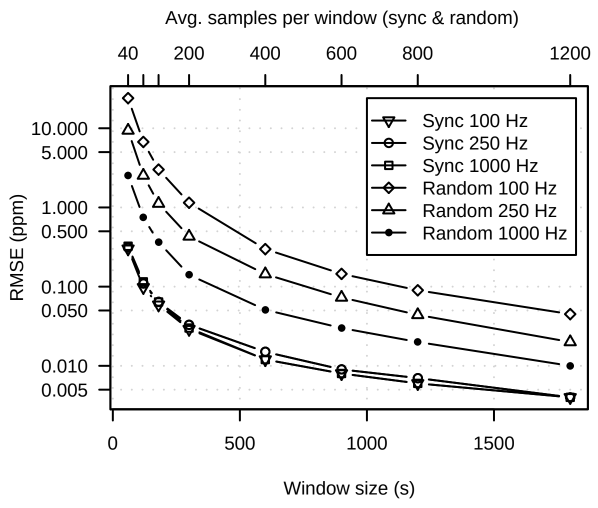


Figure 6.8: RMSEs of synchronised and random sampling for different clock frequencies of 100 Hz, 250 Hz and 1 kHz against the same target, different clock frequencies were obtained by rounding the target's timestamps (log *y*-axis)

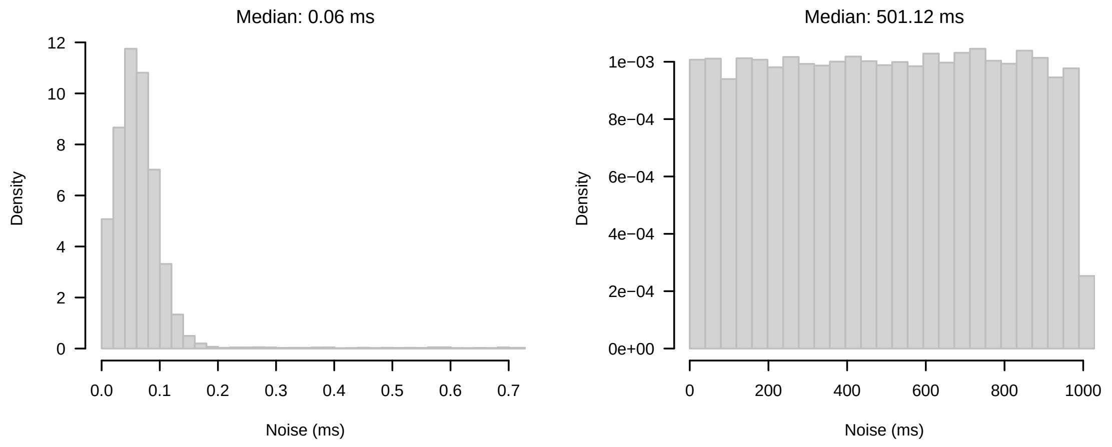


Figure 6.9: Noise distributions for HTTP probing in LAN: synchronised sampling (left) and random sampling (right)

synchronised sampling and random sampling. For synchronised sampling the noise band is only slightly larger than in Figure 6.6 because of the higher jitter. For random sampling the noise band is 1 s because of the HTTP clock frequency of 1 Hz.

Figure 6.10 shows the RMSEs of synchronised sampling, random sampling and the UDP measurement against the reference at maximum window size. The RMSEs of synchronised sampling and UDP reference are very similar to the results in Figure 6.7. Because of the large noise band, the RMSE for random sampling is more than two orders of magnitude above the RMSE for synchronised sampling. This demonstrates that our new technique is able to measure clock-skew changes for low-frequency clocks effectively, an infeasible task for random sampling.

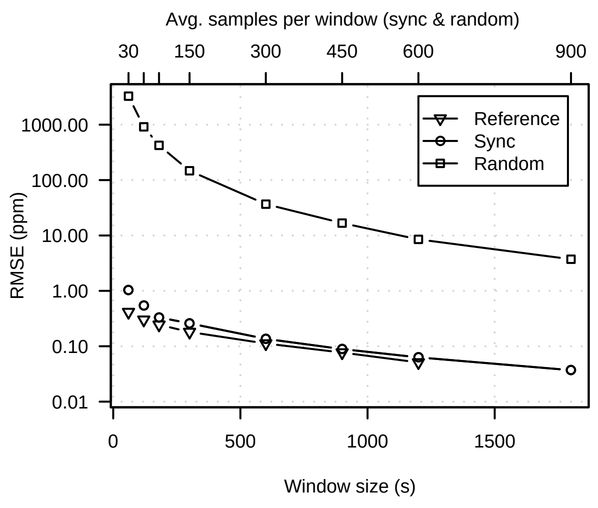


Figure 6.10: RMSEs of synchronised sampling, random sampling and the UDP measurement for HTTP probing in LAN (log y-axis)

6.4.2 Synchronised and random sampling across Internet

The attacker was the same machine as in Section 6.4.1 located in Cambridge, UK. The new target[9] had a TCP timestamp frequency of 1 kHz and was 22 hops away located in Canberra, Australia. The average RTT between measurer and target was 325 ms. We performed synchronised, random and UDP probing for approximately 21 hours, using the same parameters as in Section 6.4.1

Despite the high RTT, the jitter was relatively small and skewed towards zero (see Appendix E.2). Figure 6.11 shows the histograms of the noise bands. For synchronised sampling most of the offsets are within 250 µs of the reference whereas for random sampling there is the expected 1 ms noise band.

Figure 6.12 shows the RMSEs of synchronised sampling, random sampling and the UDP reference against the UDP reference at maximum window size. Here, the gain of synchronised sampling is smaller because of the higher network jitter, but it is still significant for smaller window sizes.

6.4.3 Attacking Tor hidden services

For our measurements we used a private Tor network. Our Tor nodes were distributed across the Internet running on PlanetLab [228] nodes. The main reason for using a private Tor network was the poor performance of hidden services in the public Tor network. Besides huge network jitter preventing accurate clock-skew measurements, hidden services always disappeared after a few hours preventing longer measurements. While currently it

[9]A PC with 2.4 GHz Celeron CPU running FreeBSD 4.10.

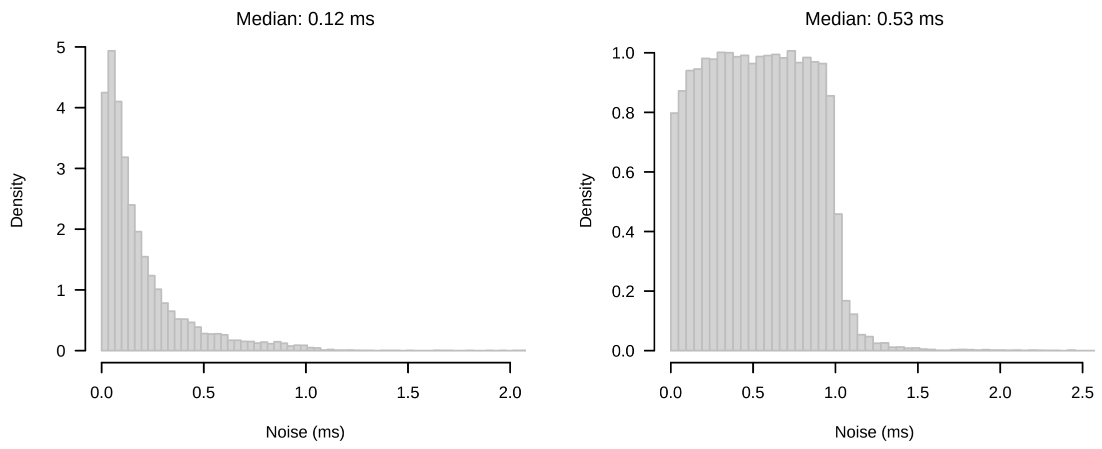


Figure 6.11: Noise distributions for Internet path: synchronised sampling (left) and random sampling (right)

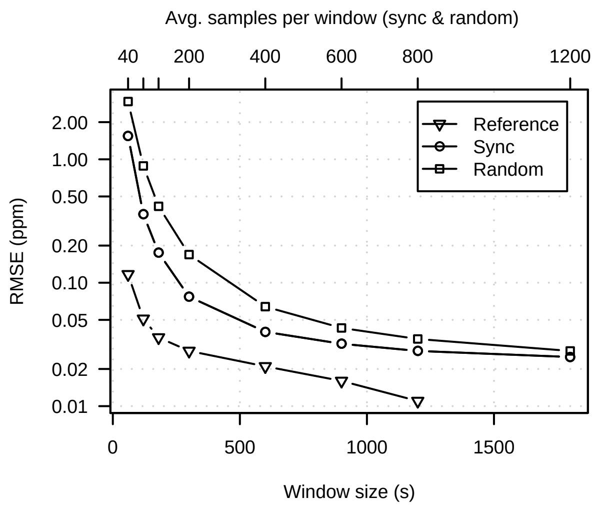


Figure 6.12: RMSEs of synchronised sampling and random sampling for TCP probing across Internet path (log *y*-axis)

is difficult to carry out the attack in the public Tor network, it should become easier in the future, since the Tor team is improving the performance of hidden services.

We selected 18 widely geographically distributed PlanetLab nodes on which we ran Tor nodes (three were directory authorities). We selected nodes that had low CPU utilisation at the time of selection. One machine hosted another Tor node and the hidden web server[10]. Another machine hosted a Tor client and our probe tool[11]. No load was induced on the server, so any clock skew changes were caused by ambient temperature changes.

First we performed an experiment similar to the ones in Section 6.4.1 and Section 6.4.2. Synchronised and random sampling were performed across the Tor network, while

[10]An Intel Core2 2.4 GHz with 4 GB RAM running Linux 2.6.16.

[11]An Intel Celeron 2.4 GHz with 1.2 GB of RAM running Linux 2.6.16. We used tsocks [229] with Tor patches to enable our tool to interact with the Tor client via the SOCKS protocol.

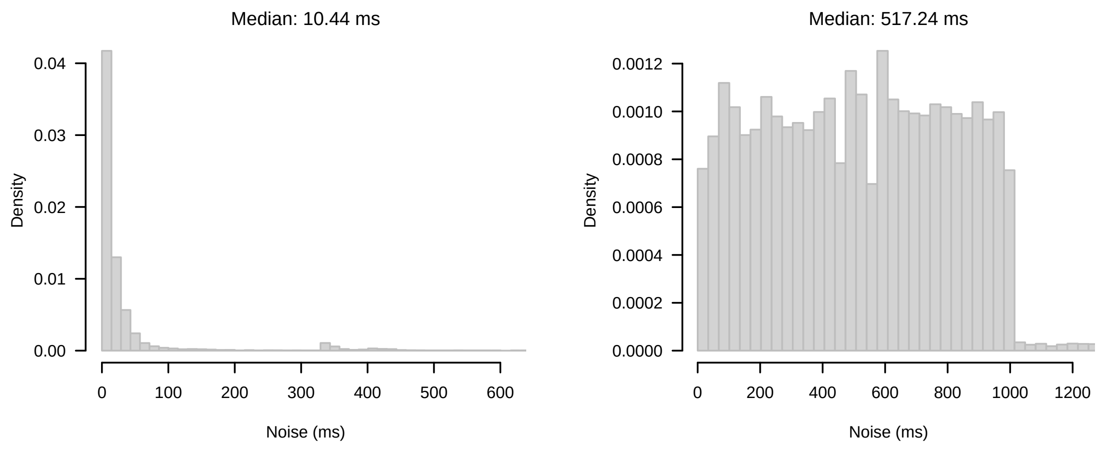


Figure 6.13: Noise distributions for private Tor testbed: synchronised sampling (left) and random sampling (right)

UDP probing was performed directly between the client and the hidden server. The measurement duration was approximately 18 hours.

The average RTT between client and hidden server across Tor was 885 ms. The RTT/2 jitter was considerably higher than in the previous measurements (see Appendix E.2). Figure 6.13 shows histograms of the noise bands of synchronised and random sampling. For random sampling it shows the expected 1 s noise band. For synchronised sampling the noise is greatly reduced. Most of the offsets are much less than 100 ms away from the slope given by the UDP reference.

Figure 6.14 shows the variable clock skew for synchronised sampling as squares (□), random sampling as circles (○) and the UDP reference as line (−) for window sizes of 1800 s and two hours. The noise is much smaller for synchronised sampling compared to random sampling especially for 1800 s windows. For two-hour windows one can clearly see a daily temperature pattern of the reference curve, with the temperature and hence the clock skew decreasing during night hours and suddenly increasing in the morning.

The synchronised sampling curve shows the same pattern with added noise. In contrast, for random sampling the pattern is not clearly visible because of the much higher noise. An attacker could use such daily temperature patterns to estimate the geographic location of the target, as described in Section 6.3.2.

In Figure 6.15 we compare the RMSEs of synchronised sampling, random sampling and the UDP reference against the UDP reference at maximum window size. The RMSE of synchronised sampling is almost one magnitude lower than the RMSE for random sampling even for window sizes as large as two hours.

In the second experiment we performed the actual attack. We treated all 19 Tor nodes as candidates and measured their clock skew directly using TCP timestamps and synchronised sampling. At the same time we measured the clock skew of the hidden web service

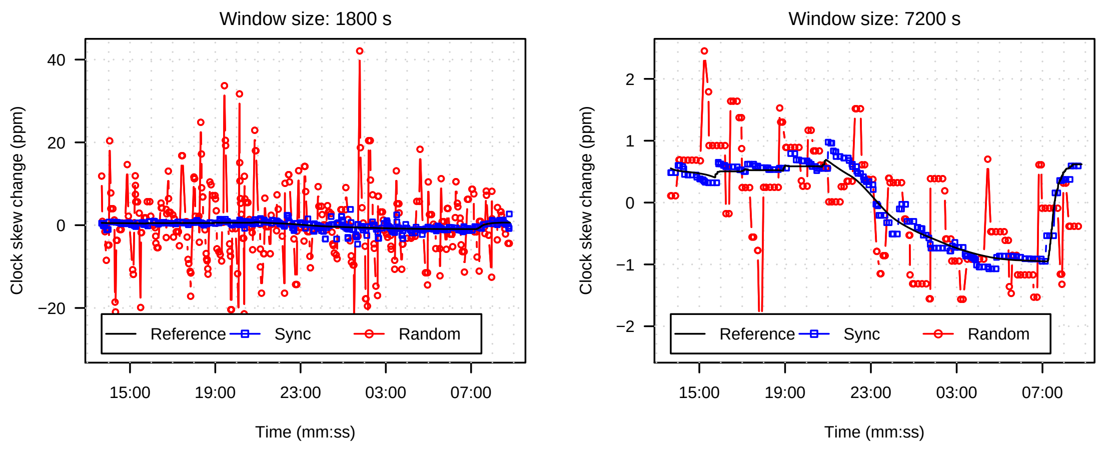


Figure 6.14: Estimated clock skew changes for hidden service in private Tor network for a window size of 1800 s (left) and 2 hours (right)

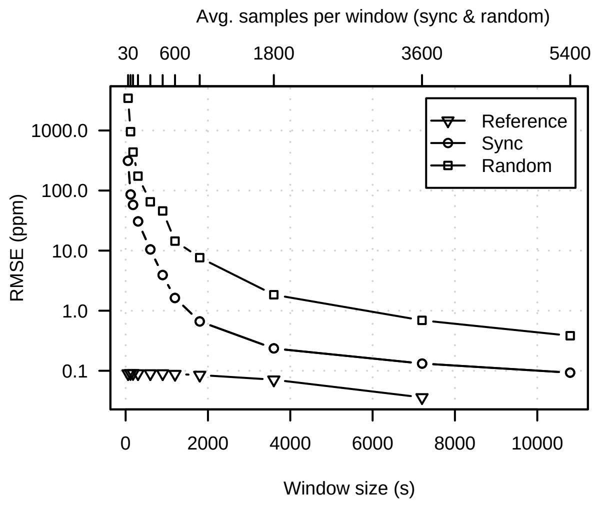


Figure 6.15: RMSEs of synchronised, random sampling and UDP reference for hidden web service in private Tor network (log *y*-axis)

via Tor based on HTTP timestamps using synchronised and random sampling simultaneously. The experiment lasted about ten hours. One of the nodes stopped responding in the middle of the experiment.

Figure 6.16 shows two graphs where each line is the RMSE between the HTTP clock skew estimate obtained from the hidden service via Tor using random/synchronised sampling and the TCP clock skew estimate of one candidate node. We used a window size of three hours, as for smaller windows random sampling was not able to consistently select one candidate as the best. The RMSE of the HTTP timestamp estimate and the correct candidate is shown as thick line while RMSEs for other candidates are shown as thin lines.

For synchronised sampling the RMSE between the Tor measurement and the direct measurement of the correct candidate is small, and with increasing duration becomes significantly smaller all other RMSEs (except one). For random sampling all RMSEs

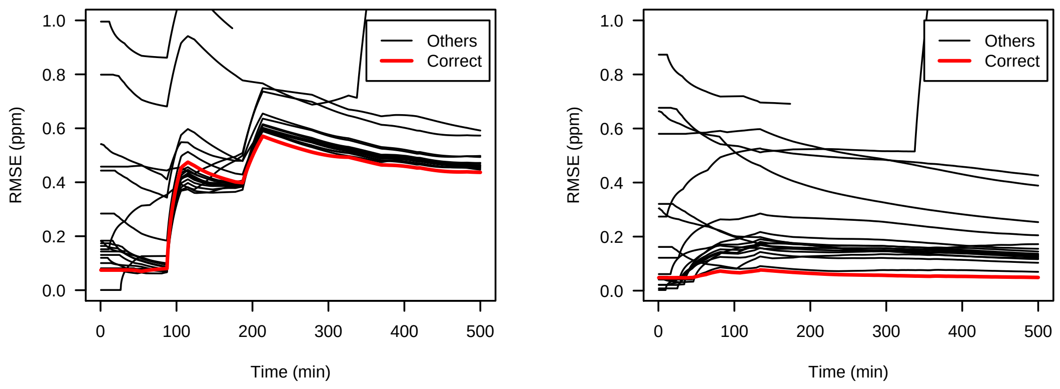


Figure 6.16: RMSEs between the HTTP clock-skew estimate obtained from hidden service via Tor using random sampling (left) or synchronised sampling (right) and the TCP clock-skew estimates of all candidate nodes

are high indicating that there is no good match of the Tor hidden service with any of the candidates. After a longer time the RMSE with the correct candidate becomes smallest, but only by a very small margin.

The time it takes to identify the hidden server is the period from the start of the measurement until the RMSE of the correct candidate becomes smallest. Note that the initial 1.5 hours it takes to get the first clock skew estimate are not included in Figure 6.16. Synchronised sampling is able to identify the correct candidate much faster than random sampling, needing only 139 minutes compared to 287 minutes.

While the variable clock skew of the TCP clock and HTTP clock are a good match, the fixed skew of the two clocks differs on our hidden server. This makes it impossible to evaluate an identification of the hidden server based on the fixed skew. However, since we know the true fixed skew of the HTTP clock, we can analyse how long it takes to get an accurate estimate using synchronised and random sampling. We use the data from the previous experiment and assume the skew estimate is correct if within 0.5 parts per million of the true value. Again synchronised sampling outperforms random sampling, needing only 23 minutes compared to 102 minutes.

6.4.4 Initial synchronisation time

We briefly analyse the initial synchronisation time, which is the time it takes until the attacker has locked on to the phase and frequency of the target's clock ticks.

Figure 6.17 plots the values of `adj_before`, `adj_behind` and `probe_interval` (see Section 6.2) over the number of samples taken from the target's HTTP clock every 2 s. The y-axis is limited to values between −10 ms and 10 ms and the x-axis is limited to the first 1 000 clock samples. Note that before-adjustments are always positive, while behind-adjustments are always negative.

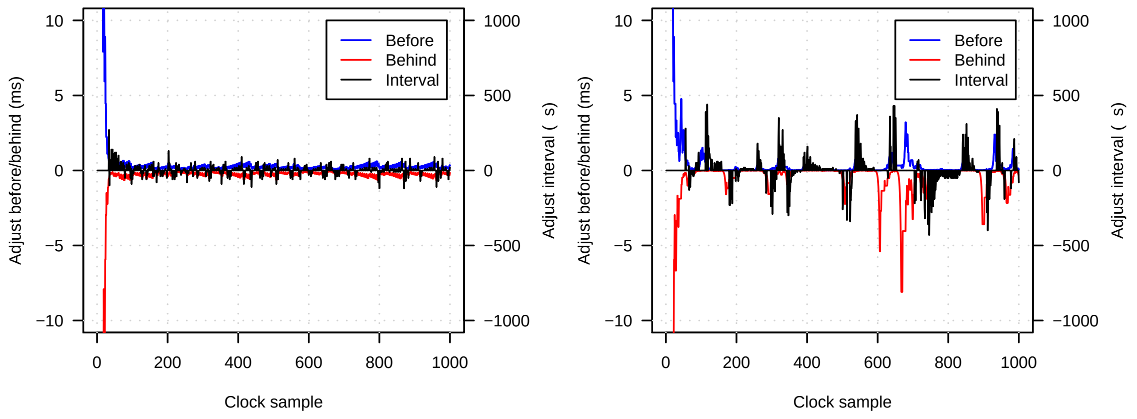


Figure 6.17: Initial synchronisation for HTTP probing in LAN (left) and probing a hidden web server over the Tor network (right)

In the LAN experiment initial synchronisation is established after only about 40 samples (approximately 1.5 minutes) and further adjustments and probe interval changes are very small (less than 500 µs and 100 µs respectively). When probing over the Tor network synchronisation is more difficult because of the much higher network jitter. Consequently initial synchronisation takes about 70 samples (approximately 2.5 minutes) and the algorithm is forced to make larger adjustments of up to several milliseconds.

6.5 Channel capacity upper bound

Now we estimate an upper bound of the capacity of the temperature-based covert channel, more specifically the improved channel with synchronised sampling.

We first propose an overall model for the channel. Our model consists of a Matlab Simulink [230] model for modelling the relation between CPU load and clock skew, and a communication channel model for modelling the transmission of information via clock-skew changes. Next, we describe a set of experiments carried out for developing the Simulink model. Then we present the Simulink model and compare its outputs with the empirical measurements. Next, we investigate the channel noise. Finally, we estimate the channel capacity for two example intermediate hosts.

6.5.1 Channel model

We model the channel as an Additive White Gaussian Noise (AWGN) channel [22], on the basis that the channel's multiple independent sources of noise are approximately Gaussian (from the Central Limit Theorem), which is confirmed by our experimental results. Thus the channel capacity depends on the channel's bandwidth and signal to noise ratio (SNR).

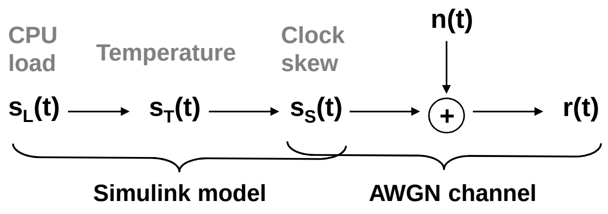


Figure 6.18: Model of the temperature-based covert channel

Figure 6.18 shows our channel model. The input of the AWGN channel is a clock skew signal generated by Alice $s_{\mathrm{S}}(t)$. The output of the channel $r(t)$ is the clock skew signal measured by Bob plus the noise. The noise $n(t)$ includes noise introduced by the clock-skew estimation (i.e. network jitter and timestamp quantisation noise) as well as noise because of CPU load or temperature fluctuations.

Alice can only indirectly manipulate $s_{\mathrm{S}}(t)$ by modulating CPU load $s_{\mathrm{L}}(t)$. To model the relationship between $s_{\mathrm{L}}(t)$ and $s_{\mathrm{S}}(t)$ through changes of temperature $s_{\mathrm{T}}(t)$ we use a Simulink model (see Section 6.5.3), parameterised according to the measured behaviour of particular intermediate hosts (see Section 6.5.2). Using the Simulink model we estimate the channel's bandwidth and signal power (see Section 6.5.5).

We estimate the noise power for a particular intermediate host from empirical measurements of $r(t)$ without input signal $s_{\mathrm{S}}(t)$. We show that the noise, after being detrended from ambient temperature trends, is indeed approximately Gaussian (see Section 6.5.4).

We estimate an upper bound of the capacity assuming ideal conditions for Alice and Bob. Alice's signal power is maximal because Alice is located on the intermediate host and CPU load is controlled directly. On the other hand the noise is minimal. The improved clock-skew estimation technique is used, and network jitter is minimal as our testbed is very small and uncongested. Ambient temperature changes are also small because all PCs were located in rooms that were air conditioned for human comfort.

6.5.2 Clock-skew measurements

We carried out several experiments for developing and fitting the Simulink model, where we varied CPU load and measured the resulting clock-skew changes.

The first intermediate host (Intermediate 1) was a 2.4 GHz Intel Celeron CPU inside a midi-tower case running Linux 2.6. Both CPU and power supply fans ran at constant speed. The second intermediate host (Intermediate 2) was a 2.8 GHz Intel Pentium CPU inside a slim desktop case running FreeBSD 4.10. It had a more effective thermally-controlled CPU fan designed so that most of the warm air is blown out of the case directly. Alice was located on the intermediate host. Bob was a second PC running Linux 2.6 connected to the same Fast Ethernet switch.

CPU load was induced using cpuburn [231]. During the experiments no network traffic, besides Bob's clock probes, was send to the intermediate host. Also, no other processes ran that used significant amounts of CPU time[12].

In our experiments we generated periodic square wave signals $s_L(t)$ and remotely measured the resulting $s_S(t)$. Each signal period was a time of maximum induced CPU load (approximately 100% load) followed by the same time of idle CPU (approximately 0% load) allowing the system to cool down to its previously unloaded temperature. Each experiment consisted of ten consecutive signal periods. We ran separate experiments using load-inducement times of 180, 300, 600, 1200, 1800, 2400 and 3600 seconds.

Changes in the ambient temperature or humidity affect the clock crystal and hence introduce noise. During our experiments we tried to minimise this noise. PCs were located in air-conditioned rooms and we performed measurements during times when doors and windows were mostly closed and no humans were inside the rooms. Nevertheless, there were some changes in the ambient conditions. However, these ambient changes usually happen on longer timescales and it is possible to remove long-term trends. Furthermore, if the induced clock skew change is large the noise is much smaller than the signal, which is the case for Intermediate 1.

We probed the TCP clock of the intermediate hosts with an average frequency of 1 Hz. Since we used synchronised sampling the timestamp quantisation noise was very low despite the low clock frequencies of 250 Hz (Intermediate 1) and 100 Hz (Intermediate 2). Connecting Alice and Bob to the same switch minimised network jitter.

With our settings one clock-skew estimate is obtained for time windows of w seconds, containing w clock samples. If w is set too small, the clock skew estimates contain a lot of noise. On the other hand too large w lead to averaging and prevent the accurate measurement of steep changes. Figure 6.19 shows the estimated variable clock skew for 3600 s load inducement on Intermediate 1 for $w = 120$ s and $w = 600$ s (average over the ten periods). The figure clearly shows the higher noise with $w = 120$ s. On the other hand the averaging caused by $w = 600$ s is also clearly visible when looking at the steep clock-skew decrease occurring when the load inducement ends. This averaging is more problematic for shorter load-inducements.

We examined window sizes of 60, 120, 180, 240, 300 and 600 seconds. We found that for 60 s and 120 s windows the noise is very high whereas for windows of 240 s and larger there is too much averaging. Therefore, for Intermediate 1 we selected $w = 180$ s. However, for Intermediate 2 the induced clock-skew changes are almost one magnitude smaller. To limit the noise to acceptable levels we used $w = 600$ s. Oversampling was used to obtain one clock-skew estimate every 30 s regardless of w.

[12]Housekeeping processes used minimal CPU resources and could have caused some network traffic.

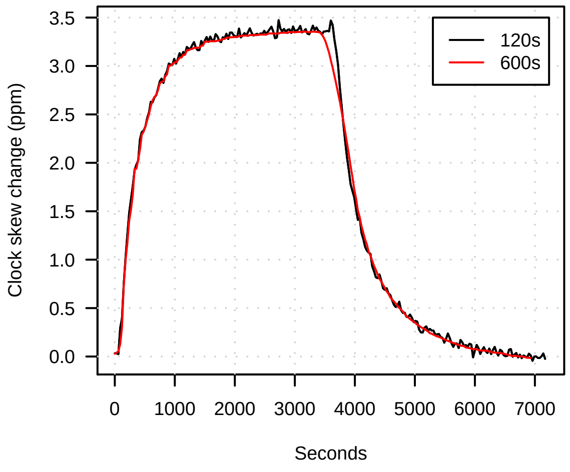


Figure 6.19: Effect of clock-skew estimation window size on the shape of the received signal

6.5.3 Simulink model

The input of the model is a CPU load signal $s_L(t)$ (values ranging from 0 to 1) and the output of the model is the corresponding clock-skew signal $s_S(t)$ (values in parts per million). Our model has two heat capacitors. One has a larger capacity and heats up slower (presumably the inside of the PC's case) while the other has a smaller capacity and heats up quicker (presumably the CPU and heat sink).

Figure 6.20 shows the model for Intermediate 1[13]. The gain constants and capacities were fitted based on the empirical data. However, the structure of the model is generic. For Intermediate 2 we used the same model but with different parameters. We believe the model could be applied to other potential intermediate hosts, as the overall shape of CPU load induced clock-skew changes looks similar for other PCs [44].

We used the same CPU load signals used in the empirical measurements as input for the model. Figure 6.21 compares the clock-skew change experimentally measured (average over all ten signal periods) with the output of the model. Overall there is a very good match. Despite the larger window size the empirical data for Intermediate 2 is much noisier given that the clock-skew changes are almost one magnitude smaller. Note that all clock-skew estimates have been detrended from ambient temperature changes and normalised to allow direct comparison. Hence Figure 6.21 shows relative changes of clock skew rather than the real absolute values.

With ambient changes removed our system is basically time-invariant as $s_S(t)$ depends only on $s_L(t)$. Ideally our system would also be linear, as a linear time invariant system is easy to analyse. Unfortunately our Simulink model shows non-linear behaviour.

[13]The capacities are $C_1 = 2.625^{-6}$ and $C_2 = 1.675^{-6}$.

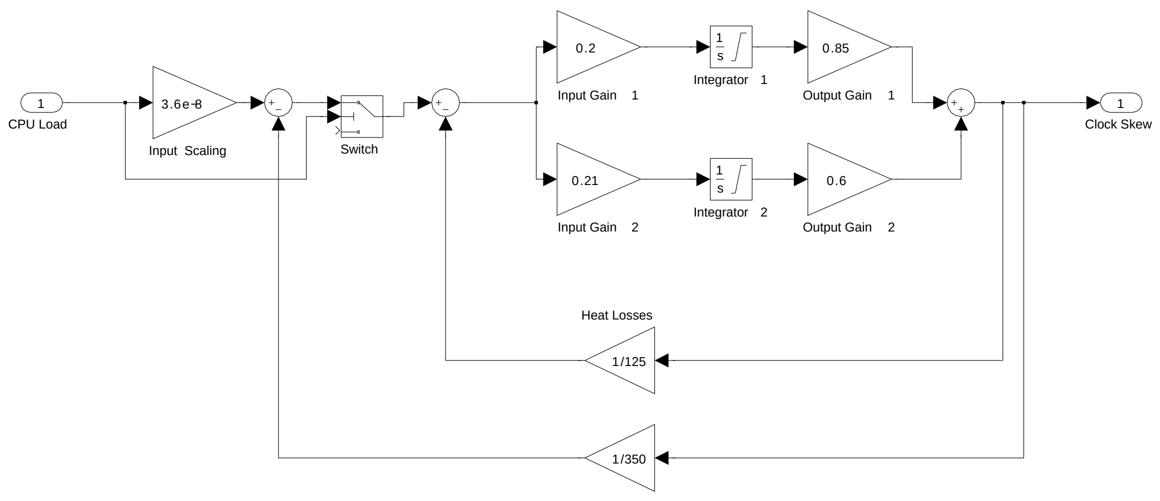


Figure 6.20: Matlab Simulink model for simulating the relationship between CPU load and clock skew (parameterised for intermediate host 1)

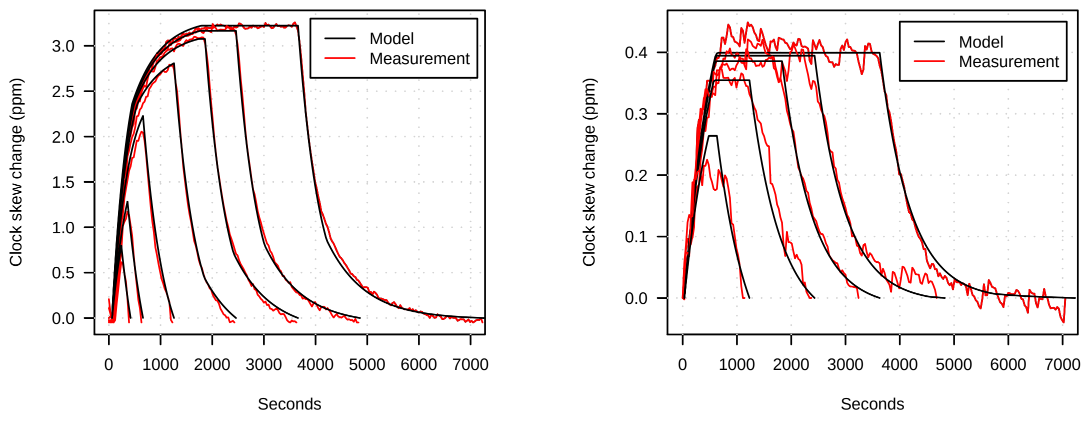


Figure 6.21: Comparison of clock-skew output of Simulink model and normalised empirical measurements for intermediate host 1 (left) and intermediate host 2 (right)

Key sources of non-linearity are the thermally controlled CPU fan (Intermediate 2 only), which results in a very quick settling of the temperature compared to a constant-speed fan. For both intermediates the cooling down is slower than the heating up because loading the CPU introduces additional energy, but when the CPU is idle there is no additional energy introduced for cooling; the thermally-controlled fan returns to its lowest speed immediately after the load inducement stops.

There are other dependencies that are non-linear in general, but within our operating conditions we assume them to be linear. In general temperature does not change linearly with CPU load, but it depends on the mix of instructions executed as well as possible CPU frequency changes. Since our CPUs ran with constant frequency and we always generated load with cpuburn we assume this relation is approximately linear. In general clock skew

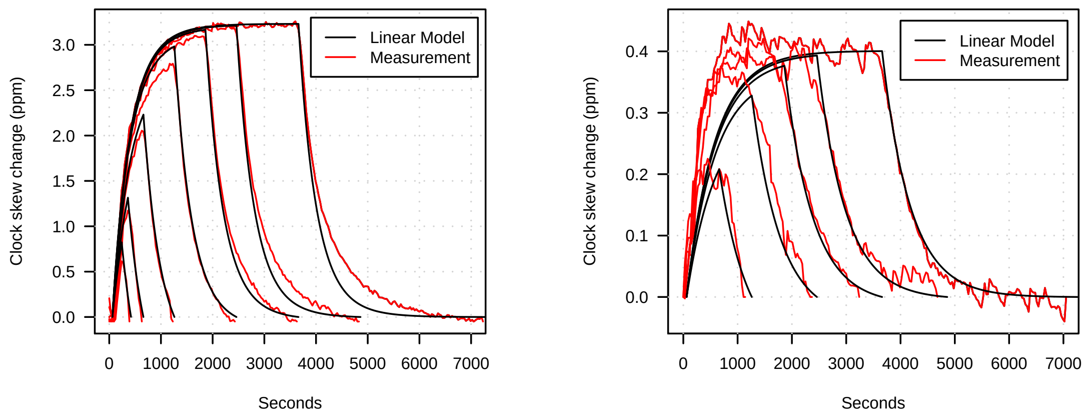


Figure 6.22: Comparison of clock-skew output of *linearised* Simulink model and normalised empirical measurements for intermediate host 1 (left) and intermediate host 2 (right)

does not change linearly with CPU load, but for typical temperatures inside PC cases the relation is also approximately linear [44].

For each intermediate host we generated a linear model from the non-linear model using the Matlab Simulink function `linmod()`[14]. Figure 6.22 compares the output of the linearised model with the detrended normalised empirical data (average over all ten signal periods). For Intermediate 1 the linear model matches quite well, although it does deviate slightly in the cool-down phase. For Intermediate 2 the linear model does not match as well because it cannot capture the very steep temperature rise and settling.

6.5.4 Channel noise

In order to estimate the noise $n(t)$ we measured clock-skew changes of the intermediate hosts, but without any CPU load inducement. We also measured the temperature inside the room and inside the PC case.

In the following graphs we show the temperatures (both normalised on the minimum temperature of each series) and the remotely measured variable clock skew for Intermediate 1. Figure 6.23(left) shows a few hours in the afternoon and evening. The case temperature is fluctuating within a few 0.1 degrees Celsius without a clear trend and does not closely follow the room temperature trend before 20:00 hours. Overall the variable clock skew looks similar to random noise.

Figure 6.23(right) shows 8–9 hours during the night. Room and case temperature are decreasing and the clock skew is decreasing accordingly. Thus the variable clock skew is not random but has a clear trend following the trend of the ambient temperature.

[14]Linearisation for individual blocks based on pre-programmed analytic block Jacobians.

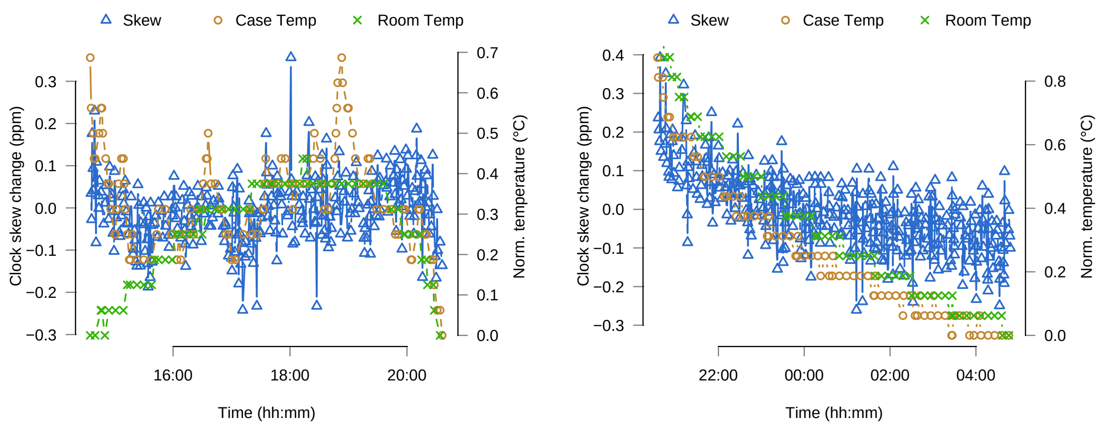


Figure 6.23: Variable clock skew and case/room temperature of intermediate host 1 during afternoon/evening (left) and night (right)

During the day temperature changes inside the case were not highly correlated with the room temperature changes. This was probably because the intermediate host PC was located in close proximity to two other PCs that were actively used during the day. During the night when all PCs were idle, the case temperature followed the trend of the room temperature closely, but was about 3.2 degrees Celsius higher. For Intermediate 2 we observed similar behaviours.

The noise is clearly not Gaussian because of the ambient changes. However, we can detrend the data from the ambient changes. We used a LOWESS smoother [232] to compute a smoothed series of data points. We then subtracted the smoothed series from the actual data series to compute the detrended series. In the following we investigate whether the detrended noise has a Gaussian distribution.

We use the Shapiro-Wilk statistical test of normality (see Appendix E.3). For Intermediate 1 with outliers removed we cannot reject the hypothesis that the data is Normally distributed at 99% significance level. For Intermediate 2 we cannot draw the same conclusion. However, the statistical test is quite sensitive to small deviations.

Figure 6.24 and Figure 6.25 show quantile-quantile (QQ) plots of the empirical distributions against the theoretical Normal distribution for both intermediate hosts. In all graphs the points follow the quantile-quantile line closely, except at the edges. This indicates that the empirical distributions are roughly Gaussian, except for some outliers.

6.5.5 Channel capacity

The channel is time-invariant, and we showed that the noise is approximately Gaussian. Therefore, we model the covert channel as an AWGN channel. The channel capacity is:

$$C = B \cdot \log_2\left(1 + \frac{P}{N}\right), \tag{6.3}$$

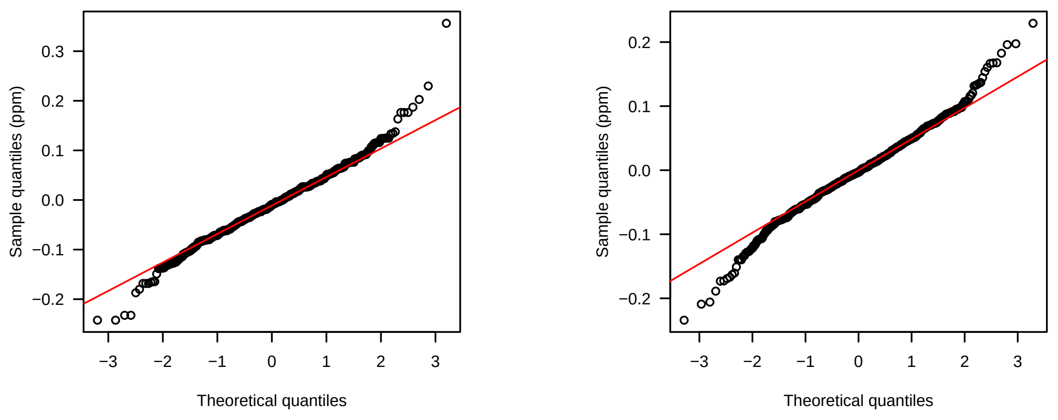


Figure 6.24: QQ plots of detrended variable clock skew during the day (left) and night (right) for intermediate host 1

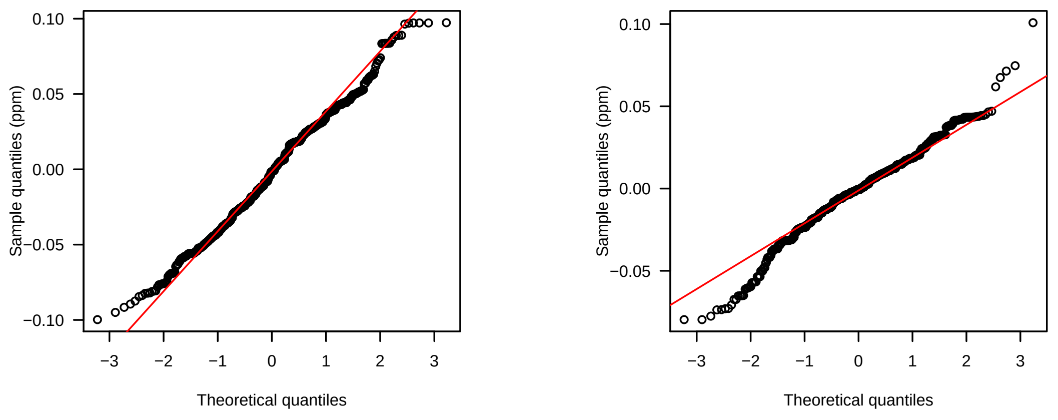


Figure 6.25: QQ plots of detrended variable clock skew during the day (left) and night (right) for intermediate host 2

where B is the bandwidth of the channel, P is the average signal power and N is the average noise power [22].

The covert channel is basically a base-band system acting as a low-pass filter on the input signal. To estimate the bandwidth we need to estimate the upper cut-off frequency, which is commonly defined as the frequency where the power of the output has decreased by 3 decibel (dB). For Intermediate 1 we computed the bandwidth directly from the linear Simulink model (Bode plot). For Intermediate 2 the linear model does not fit very well. We estimated the bandwidth by simulating different signal period lengths with the Simulink model and identifying when the power has decreased by approximately 3 dB.

The bandwidths are $B_1 \approx 0.000434$ Hz for Intermediate 1 and $B_2 \approx 0.000444$ Hz for Intermediate 2. This means the period of the signal is approximately 2304 s (Intermediate 1) and 2250 s (Intermediate 2), which is equivalent to 1152 s load followed by 1152 s idle time and 1125 s load followed by 1125 s idle time. This is broadly consistent with the results from the empirical measurements (see Figure 6.21). For Intermediate 1 the

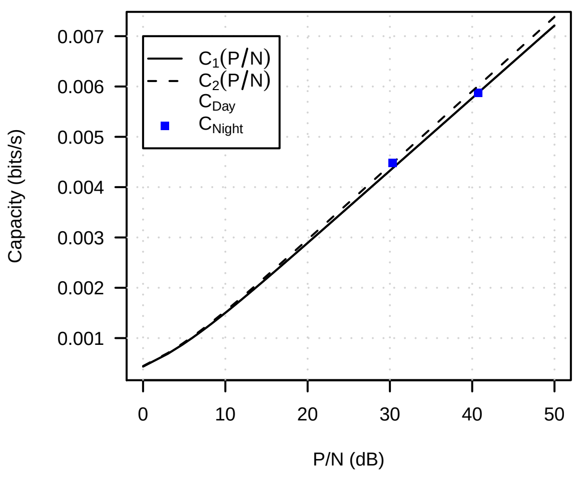


Figure 6.26: Channel capacity for both intermediate hosts based on signal-to-noise ratio, where the points C_{Day} and C_{Night} depict the capacities given the empirically measured channel noise during the day and night on each capacity curve

measured 1200 s load inducement signal has approximately 52% of the power, which is exactly what the model predicts. For Intermediate 2 the measured 1200 s load inducement signal has approximately 47% of the power. The model predicts approximately 53%, but it is hard to empirically measure the low signal power exactly.

Figure 6.26 shows the channel capacity based on the SNR in dB for Intermediate 1 (C_1) and Intermediate 2 (C_2). The capacity increases almost linearly with the SNR for larger SNRs. However, the sending power is not unlimited and hence the capacity cannot increase to infinity. The question is: what SNRs can be achieved?

We estimated the average power of signal and noise by computing the power spectral density (power per frequency band), integrating over all frequency bands within the channel bandwidth and then normalising the power based on the number of samples of the signal. We computed the average signal power based on the Simulink model output for alternating 0% and 100% CPU load with a frequency equal to the channel bandwidth. We computed the average noise power from the empirically measured detrended noise signals, separately for day and night time.

As expected for both intermediates the SNRs are higher during the night, as there was more noise during the day. For Intermediate 2 the SNRs are smaller, despite lower noise due to the larger sample windows, because of the much smaller signal power. Figure 6.26 shows the capacities C_{Day} and C_{Night} for the day and night SNRs on both capacity curves.

Depending on the noise the capacity is between 0.0046 bits/s and 0.0059 bits/s (Intermediate 1) and between 0.0032 bits/s and 0.0045 bits/s (Intermediate 2). This equates to around 16.4–21.1 and 11.7–16.1 bits per hour. Note that these estimates are upper bounds, because we assumed the noise power to be minimal (idle intermediate host) and the signal power to be maximal (100% CPU load with cpuburn). In reality the capacity is likely to be lower, because the CPU load that the covert sender can generate is smaller, and the noise on the channel is larger (higher CPU load jitter and network jitter).

6.5.6 Discussion

Our capacity estimates are significantly higher than Murdoch's estimate of 2–8 bits per hour [44]. However, this estimate was ad-hoc based on an inspection of the experimental data and could not consider the improved clock-skew measurement technique.

The low capacity makes the channel less relevant for general-purpose communication, since there exist other covert channels with much higher capacities, as shown previously. However, there may be situations in which this is the only available channel, because many other channels are easy to eliminate. Furthermore, the temperature-based covert channel can penetrate "air-gap" security boundaries [44]. It can be used across devices not connected by a network, if the devices can have temperature effects on each other.

The capacity of the channel is more than sufficient for revealing hidden servers since in this scenario only a few bits need to be transmitted. The probability of choosing a wrong host from the candidate set, a false positive, is $p_{\mathrm{FP}} = 2^{-n}$ where n is the number of covert bits transmitted [44]. For example, even if only 16 bits are transmitted the probability of a false positive is only $p_{\mathrm{FP}} = 1.525879^{-5}$. Given our capacity estimates it takes only 1–2 hours to transmit 16 bits.

Covert channels with capacities of less than one bit per second are often deemed acceptable [19]. However, in scenarios where small capacities pose a security threat, such as in anonymisation networks, temperature-based covert channels require handling.

6.6 Conclusions

Temperature-based covert channels suffer from two main sources of noise: timestamp quantisation noise and network jitter. Timestamp quantisation noise is often much worse than network jitter, especially if the intermediate host's accessible clock has a low frequency and the network path between covert sender and receiver is uncongested.

We developed an enhanced covert channel by improving the remote clock-skew estimation based on Murdoch's idea of synchronised sampling [44]. The evaluation shows that our new algorithm provides more accurate clock-skew estimates than the previous approach. The quantisation noise is significantly reduced to a small margin independent of the clock frequency. If the clock frequency is low, the accuracy improves by up to two orders of magnitude. This means the channel capacity is increased.

The improved covert channel not only increases the efficiency of Murdoch's attack against hidden services [44]. It also paves the way for new more efficient attacks against hidden servers generating a lot less network traffic, which we described and partly evaluated. Our improved clock-skew estimation technique could also be used to improve the identification of hosts based on their clock skew as proposed in [221]. An open-source proof-of-concept implementation of our novel clock-skew measurement technique has been released [233].

We also proposed a method for estimating the capacity of temperature-based covert channels. Based on empirical measurements we developed a Simulink model for modelling the relation between CPU load and clock-skew, and used it to determine the channel's bandwidth and signal power. We showed that the measured detrended noise is approximately Gaussian. Therefore, we estimated the capacity based on the additive white Gaussian noise channel model. For two example intermediate hosts we showed that upper bounds of the capacity are around 11.7–16.1 and 16.4–21.1 bits per hour.

The capacity of the channel is more than sufficient for attacking hidden servers. But for general-purpose communications it is quite small, and there are other covert channels with higher capacities. However, many other channels are easy to eliminate and there may be situations where the temperature-based covert channel is the only usable channel.

6.6.1 Future work

We showed that our new proposed attacks against Tor hidden services work in principle, but we did not provide a comprehensive evaluation and the private Tor network we used was relatively small. Future work could extend our evaluation and analyse the sensitivity and specificity of our new attacks using a larger test network with more hidden servers or possibly even the public Tor network itself.

Our proof-of-concept implementation runs in userspace, which naturally limits its ability to exactly time probe packets. A kernel implementation, use of network cards capable of high-precision traffic generation, or use of a real-time kernel could further increase the accuracy. Furthermore, our implementation could be improved by fine-tuning the algorithm parameters.

We did not investigate suitable encoding techniques. Our Simulink model in combination with the Matlab Simulink communications toolbox can be used to develop encoding and error correction techniques, and measure the actual throughput of the channel. The development and analysis of techniques to use remote clock-skew estimation for approximate geo-location is an interesting subject left for further study.

With respect to the estimation of the channel capacity, future work could examine a larger number of different intermediate hosts and estimate the channel capacity when CPU load is generated remotely through varying the rate of requests send across the network to typical server applications (e.g. web server).

Chapter 7

Countermeasures

In the previous chapters we analysed the performance of different covert channels and showed that, apart from the temperature-based covert channel, channel capacities are sufficient for general-purpose communication. We also demonstrated that even with non-optimal encoding schemes the achievable throughput is reasonably high for practical use, even in the presence of moderate channel errors. We now develop and analyse techniques for detecting and eliminating the covert channels.

The TTL covert channel is not hard to eliminate in principle, but simple approaches may not be efficient. We propose a method that has no negative impact on the network traffic and is efficient enough for real-world deployment. The inter-packet timing channel cannot be eliminated completely, but its capacity can be drastically reduced by introducing artificial network jitter. We show that this method practically eliminates the channel without having a larger negative impact on the performance of the applications that generate the overt traffic.

The multiplayer game traffic channel can only be eliminated by eliminating the overt traffic (the game). It is hard to completely eliminate the temperature-based covert channel. However, given that the channel's capacity is very small, it only requires handling in certain scenarios, such as anonymisation networks. Furthermore, we propose measures to further reduce the channel's capacity.

We evaluate the use of Machine Learning (ML) techniques for detecting different types of covert channels. First we describe our ML-based approach, and then apply it to detect the different covert channels. We show that ML techniques are successful in detecting TTL and game-traffic covert channels with over 95% accuracy. If inter-packet times are auto-correlated the timing channel proposed in [104, 105] is also detected with over 95% accuracy. However, the new improved timing channel is harder to detect with accuracies of only up to 70–80%.

Because the channel capacity of the temperature-based channel is very low and we propose measures to further reduce it, we do not include this channel in our analysis.

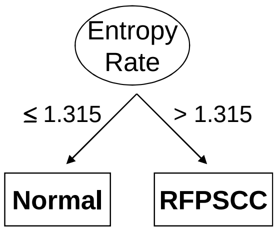


Figure 7.1: Example classifier built by C4.5 that detects the RFPSCC covert channel based on the estimated entropy rate of modulo delta pitch angle changes

7.1 Machine Learning

We use supervised ML to differentiate between normal traffic and covert channels. Supervised ML techniques build a classifier in the training phase based on data instances with class labels attached. The classifier is build so that the data instances are ‘optimally’ separated into the different classes based on characteristics (features) of the instances other than the class label. All better ML techniques build ‘optimal’ classifiers, but avoid making the classifier too specific (avoid overfitting). The classifier is then used in the testing phase to classify data instances of unknown class.

There are many different ML algorithms [234]. Previous research showed that for classification of network traffic the better techniques provide similar accuracy, but differ greatly regarding training time and classification speed [235]. We use the C4.5 decision tree classifier [236], more precisely its implementation in the Waikato Environment for Knowledge Analysis (WEKA) [237], because it performed well previously [235]. Using a decision tree algorithm has the advantage that a human can interpret the resulting classifier (classification tree), although with increasing size this becomes difficult.

7.1.1 C4.5 decision tree

C4.5 creates a classifier based on a tree structure of nodes, branches and leaves [236]. Nodes in the tree represent features, and branches represent value tests. A series of nodes and branches is terminated by a leaf, which represents the class. Determining the class of an instance is simply a matter of tracing the path of nodes and branches to a terminating leaf node. The complexity of a tree typically increases with increasing number of independent features and increasing size of the training data.

Figure 7.1 illustrates a small classification tree. This particular classifier was generated by C4.5 to detect RFPSCC based on the estimated entropy rate of modulo delta pitch angle changes (see Section 7.6).

C4.5, as other decision tree learners, uses the ‘divide and conquer’ method to construct a tree from a set of training instances S. If all cases in S belong to the same class, the decision tree is a leaf labelled with that class. Otherwise the algorithm will use tests to divide S into several non-trivial partitions.

Each of the partitions becomes a child node of the current node and the tests to separate S are assigned to the branches. C4.5 uses two types of tests each involving only a single attribute A. In case of discrete attributes the test is $A = ?$ with one outcome for each value of A. For real attributes the test is $A \leq \theta$ where θ is a constant threshold. To find the optimal partitions C4.5 relies on greedy search and selects the test set that maximizes an entropy-based gain ratio [236].

The divide and conquer approach partitions the data until every leaf contains instances from only one class or a further partition is not possible because two instances have the same features but different class. If there are no conflicting cases the tree will correctly classify all training instances. This over-fitting leads to a decrease of the prediction accuracy. C4.5 attempts to avoid over-fitting by removing some structure from the tree after it has been built (tree pruning) [236].

Because C4.5 selects the feature tests in order of maximising the entropy-based gain ratio it is not adversely affected by unimportant or irrelevant features like some other techniques. The most useful features are always used at the top of the tree and irrelevant features are ignored. Using a feature selection technique is not necessary, although sometimes it still improves accuracy slightly [235]. In our experiments we used the default WEKA 3.4.4 settings for all parameters of C4.5.

7.1.2 Evaluation metrics

To determine the accuracy we compute the true positive rate, false positive rate, precision and recall for each class (covert and normal). A true positive (TP) is a class instance correctly classified. A false positive (FP) is a non-class member misclassified as class member and a false negative (FN) is a class member misclassified as non-class member.

Recall or *TP rate* is the number of class members classified correctly over the total number of class members:

$$\text{recall} = \text{TP rate} = \frac{\text{TP}}{\text{TP} + \text{FN}} . \tag{7.1}$$

The *overall TP rate* is the total number of correctly classified instances divided by the total number of instances, which is equivalent to the average TP rate over both classes. The *FP rate* is the number of FPs over the total number of class members:

$$\text{FP rate} = \frac{\text{FP}}{\text{TP} + \text{FN}} . \tag{7.2}$$

Precision is the number of class members classified correctly over the total number of instances classified as class members:

$$\text{precision} = \frac{\text{TP}}{\text{TP}+\text{FP}} \ . \tag{7.3}$$

The *F-measure* is the evenly weighted harmonic mean of precision and recall [237]:

$$F = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \ . \tag{7.4}$$

For recall, precision and F-measure higher values are better. For FP rate lower values are better. Since there are two classes and we usually use classes of same size, a precision, recall or F-measure of 50% is equal to random guessing.

We also examine the complexity of a classifier. As measure of complexity we use the total number of nodes of the resulting decision tree. The smaller the complexity is, the smaller the memory footprint and classification time of the classifier are[1].

We perform k-fold cross-validation for each dataset. The data set is randomly divided into k subsets, and the classification is repeated k times. Each time, one of the k subsets is used as the test set and the other $k-1$ subsets form the training set. Then the average accuracy statistics across all k trials are computed. The advantage of this method is that less importance is placed on how the data is divided. Every data instance is in a test set exactly once, and is in a training set $k-1$ times. The variance of the accuracy estimate is reduced as k is increased. We always perform 10-fold cross-validation.

7.1.3 Generic features

Here we describe generic features used for detecting different channels. Features developed for specific channels are described in the respective sections of each channel. For generic features that have parameters we describe the choice of parameter values in the respective sections of each channel.

Let $X = [X_1, \ldots, X_n]$ be a series of random variables with values $x_1, \ldots, x_n$. Two features we use for all channels are the first order entropy (referred to as *Entropy*) and an estimate of the entropy rate (referred to as *EntropyRate*). The first order entropy (Shannon entropy) is a useful metric to compare the shape of distributions of random variables. High values indicate higher uncertainty whereas low values indicate lower uncertainty of the random variables. Entropy is defined as [22]:

$$\text{EN}(X_1, \ldots, X_m) = -\sum P(x_1, \ldots, x_m) \cdot \log_2 P(x_1, \ldots, x_m) \ . \tag{7.5}$$

For comparing the complexity/regularity of a time series of random variables we compute an estimate of the entropy rate. The entropy rate is high if the data series has high complexity; it is low if the series has regularities.

[1]WEKA includes the root node in the count, meaning an ‘empty’ tree has a size of one.

The exact entropy rate of a finite sequence of observations cannot be measured and must be estimated. To estimate the entropy rate we use the corrected conditional entropy (CCE) [238]. The conditional entropy (CE) of X_m given the previously observed sequence $X_1, \ldots, X_{m-1}$ is:

$$\mathrm{CE}\left(X_m | X_1 \ldots X_{m-1}\right) = \mathrm{EN}\left(X_1, \ldots, X_m\right) - \mathrm{EN}\left(X_1, \ldots, X_{m-1}\right) \ . \tag{7.6}$$

The CCE is defined as:

$$\mathrm{CCE}\left(X_m | X_1, \ldots, X_{m-1}\right) = \mathrm{CE}\left(X_m | X_1, \ldots, X_{m-1}\right) + \mathrm{perc}\left(X_m\right) \mathrm{EN}\left(X_1\right) \ , \tag{7.7}$$

where $\mathrm{perc}\left(X_m\right)$ is the percentage of unique patterns of length m and $\mathrm{EN}\left(X_1\right)$ is the first order entropy of X_1. The estimate of the entropy rate is the minimum of CCE over different values of m:

$$\mathrm{EER}\left(X\right) = \min\left(\mathrm{CCE}\left(X, m\right) | m = 1, \ldots, n\right) \ . \tag{7.8}$$

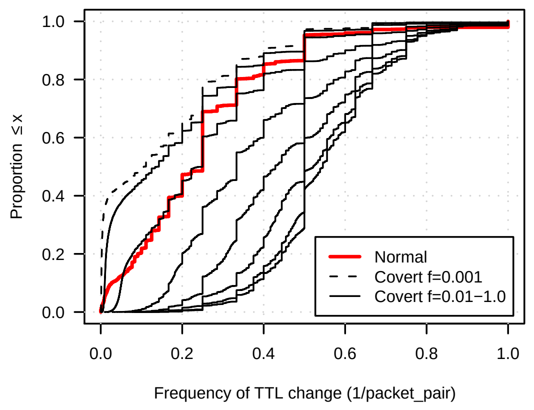
The minimum exists because CE decreases while the corrective term perc() increases with increasing m. Given our datasets the minimum is usually reached for small m and hence the search can be terminated quickly.

To compute the entropy rate, the data needs to be binned. Previous work suggested that equiprobable binning[2] of the data is the most effective [238, 107]. The number of bins Q must be chosen a-priori. A larger Q retains a larger amount of information. However, if Q is too large the number of possible patterns is increased exponentially (Q^m) and the ability to recognise longer patterns is reduced (CCE is dominated by the second term). We select Q based on some initial experiments.

7.2 Detection of TTL channels

The warden needs to distinguish between normal TTL variation and the covert channel. First, we discuss several strategies how the warden can possibly detect the covert channel. Next, we present a modified sparse encoding scheme tailored to the TTL channel that improves the stealth for smaller encoding fractions. Finally, we analyse how effectively Wendy can detect the channel on a per-flow basis. We describe the datasets and features used, and present the results of our evaluation.

[2]The size of bins is selected so that each bin holds approximately the same number of instances.



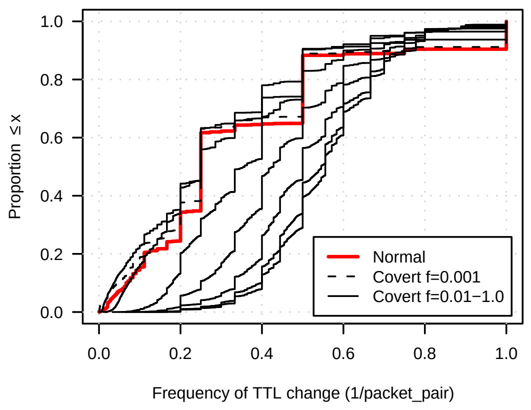


Figure 7.2: Frequency of TTL changes depending on the encoding fraction for the Twente trace (left) and the Bell trace (right)

7.2.1 Detection strategies

If Wendy is only one hop away from Alice, detection is trivial as any observed TTL variation reveals the covert channel. Otherwise, detection is harder than detection of noiseless channels, because of the presence of normal TTL variation (see Section 3.1).

The DUB modulation scheme is trivial to detect, as the number of distinct TTL values exceeds that of normal flows. The other modulation schemes are not as apparent if only one bit is encoded per TTL or TTL pair with an amplitude of one (see Section 3.1).

A high frequency of TTL changes is suspicious since without covert channel usually only a very small percentage of flows experience many TTL changes (see Section 3.1.4). However, Alice can improve the stealth through lowering the rate of induced TTL changes, and thereby also the transmission rate, by using the sparse encoding technique proposed in Section 4.2.2.

Figure 7.2 shows the CDFs of the frequency of TTL changes depending on the fraction of overt packets used (0.1%, 1%, 5%, 20%, 40%, 60%, 80%, 100%). The frequency decreases with decreasing encoding fraction. For high encoding fractions the covert channel is obvious, but for low encoding fractions the frequency of TTL changes is similar to the frequency of normal TTL variation.

Wendy can also monitor the overall percentage of flows with TTL changes. If she knows the usual percentage she can detect abnormal increases. Figure 7.3 shows the increase of the number of flows with TTL changes over the encoding fraction. For higher encoding fractions there is a huge increase for the Twente trace. However, for the Bell trace the increase is more modest because the normal TTL noise is more than one magnitude higher than in the Twente trace. When the encoding fraction is reduced to values close to the natural noise level there is no large increase.

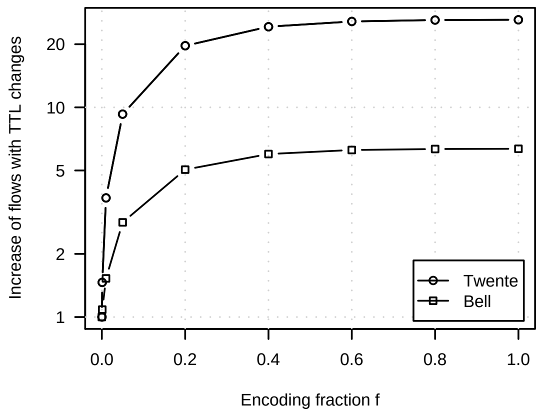


Figure 7.3: Increase in number of flows with TTL changes with increasing encoding fraction (log y-axis)

In order to achieve low TTL change frequencies Alice and Bob must drastically reduce their rate to about 5% of the maximum rate. However, even with this small encoding fraction the throughput could still be in the order of tens of bits per second.

For low encoding fractions Wendy cannot observe an increase of TTL changes. Furthermore, in reality Alice and Bob would often encode the covert channel in only a small number of flows, not affecting distributions of metrics computed for large sets of flows. Then Wendy can only detect the channel by an in-depth comparison of a flow's TTL change patterns with regular change patterns.

7.2.2 Modified sparse encoding

Sparse encoding selects overt packets uniformly (see Section 4.2.2), which is problematic for the TTL channel at very low encoding fractions. In Section 7.2.5 we show that detection accuracy decreases with decreasing encoding fraction, but then actually increases again for very small encoding fractions. This happens because the volume of the overt traffic is dominated by a small number of large flows that have many packets. For small encoding fractions most of the covert bits are encoded in these large flows. However, normal TTL variation is not biased towards large flows.

A modified sparse encoding scheme improves the stealth for very low encoding fractions. It encodes covert bits with higher probability in short flows or at the beginning of long flows. Let n be the sequence number of a packet in a flow as determined by Alice and Bob, let $n_{\max}$ be the maximum number of packets in a short flow and let R_S be the total number of packets divided by the number of packets in short flows for a traffic mix. Given a desired encoding fraction f the actual packet selection probability $\tilde{f}$ is:

$$\tilde{f} = \begin{cases} \min(1, R_S \cdot f) \ , & n \le n_{\max} \\ \frac{1}{R_S - 1} \max(0, R_S \cdot f - 1) \ , & n > n_{\max} \end{cases} . \tag{7.9}$$

Then analogue to Equation 4.7 in Section 4.2.2 packets are selected as carrier of the covert channel if ($\oplus$ operator denotes the XOR function):

$$H(b \oplus k) \le \tilde{f} \, , \tag{7.10}$$

where H is a good hash function, b is the hash input taken from a packet and k is part of Alice's and Bob's shared secret.

It is not necessary to have a very accurate estimate of R_S. However, if R_S is not accurate the desired encoding fraction is not achieved. Modified sparse encoding relies on packet counters being synchronised between Alice and Bob, for example TCP sequence numbers or timestamps. Otherwise, there are synchronisation errors if there is packet loss.

7.2.3 Datasets and methodology

As source of overt traffic we used the Twente and Bell traces described in Section 3.1 (approximately 40 million and 20 million packets). As in Section 3.1 we only considered flows with at least four packets and an average packet rate of at least one packet per second. Although this significantly reduced the number of flows, our datasets were still left with approximately 38 million and 19 million packets respectively.

We used CCHEF to encode the covert channel into a fraction of the overt traffic (0.1%, 1%, 5%, 20%, 40%, 60%, 80%, 100%) using the MED and DED modulation schemes. Packet loss and reordering were always zero. The modulated overt traffic was stored as a new trace file. For modified sparse encoding we computed R_S from the trace files and verified that $\tilde{f} \approx f$ (relative error was always below 0.5%)[3]. We then computed the features for the traces with the covert channel as well as for the original unmodified traces.

For each modulation scheme and encoding fraction we created a dataset containing the same number of normal flows and flows with covert channel (randomly sampled from the larger set). Equal-sized classes prevent a bias of the classifier towards a larger class. For Twente each dataset has 6 430 flows. For Bell each dataset has 25 990 flows due to the larger number of normal flows with TTL changes.

In our datasets the number of packets per flow (size of flows) and hence the number of covert bits per flow differs greatly. Therefore, we do not only analyse the flow-based classifier accuracy but also the bit-based classifier accuracy based on the number of TTL

[3] We used $n_{\max} = 50$, $R_S = 5.85$ (Twente) and $R_S = 7.52$ (Bell).

changes, which for the covert channel is proportional to the number of covert bits. The latter better reflects how much of the covert data is detected.

As metric for the classification accuracy (detection accuracy) we use the F-measure averaged over both classes because in terms of TTL changes the two classes are very unbalanced. Even with small encoding fractions the covert class has many more TTL changes and thus the average bit-based TP rate is completely dominated by the accuracy of the covert class. In contrast both classes contribute evenly in the averaged F-measure.

The instances of the covert class are randomly sampled from a larger data set. Furthermore, during cross-validation instances are randomly selected for either the training or the testing set. Preliminary tests showed that flow-based accuracy only varied slightly, but bit-based accuracy and complexity varied significantly. Therefore, we performed cross-validation 25 times and report average performance metrics.

7.2.4 Features

Let $X = [X_1, \ldots, X_n]$ be the series of TTLs of a flow with values $x_1, \ldots, x_n$. Since the occurrence of normal TTL changes largely depends on the traffic sources, as most changes are not routing-related (see Section 3.1), there could be correlations between the observed TTL change patterns and the actual TTL values. To eliminate possible bias we normalised all TTLs on a per-flow basis dividing each TTL by the minimum TTL:

$$X_{\text{norm}} = x_1, \ldots, x_n - \min(x_1, \ldots, x_n) \ . \tag{7.11}$$

For each flow we computed the Entropy and the EntropyRate of the TTL values. For the latter the number of bins Q was set to the number of unique values in X (usually two) or a maximum value Q_{max}, whichever was higher. We performed a number of initial experiments and found that $Q_{\text{max}} = 5$ provided the best results.

The series X_{norm} can be converted into a series of zeros and ones, where a one indicates a TTL change and a zero indicates no TTL change. This series can be viewed as a series of m binary random variables $Y = [Y_1, \ldots, Y_m]$. The next two features are based on Y.

The first feature is based on the observation that normal TTL changes often occur at the start or end of flows. We define the distribution of positions of TTL changes as:

$$\text{CPOS} = \left\lfloor \left(\frac{\text{index}(Y(y=1))}{(m-1)} \right) B \right\rfloor / B \ , \tag{7.12}$$

where $\text{index}(Y(y=1))$ are the indices of the packet pairs where changes occurred (ranging from $[0, m-1]$) and B is the number of equal-sized bins[4]. We then used the median

[4]We set $B = 10$ in our experiments.

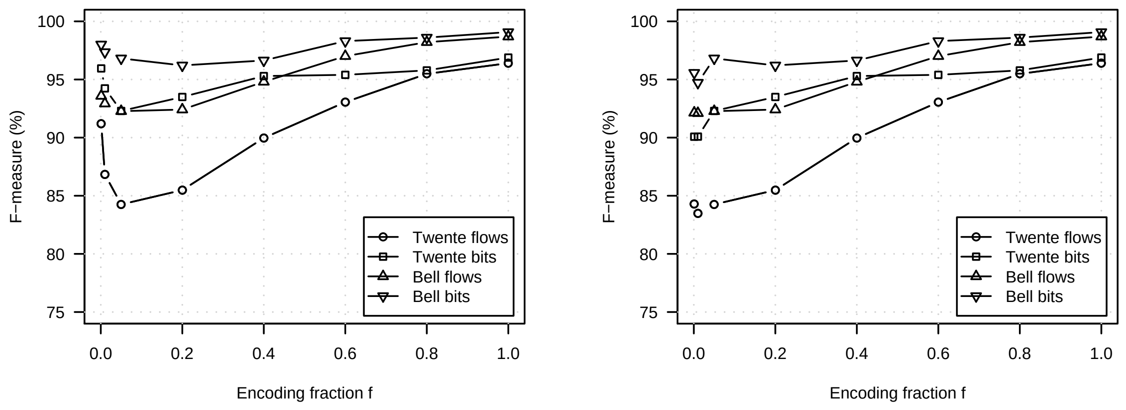


Figure 7.4: Detection accuracy for the MED modulation scheme for both traces using sparse encoding (left) and modified sparse encoding (right)

and the first-order entropy of the distribution as features (referred to as *ChangePosMedian* and *ChangePosEntropy*).

We also computed the runs test for *Y* using the R function `runs.test()` [239]. The runs test is a statistical test that tests whether a data series is random or not. The runs test statistic is used as a feature (referred to as *RunsStatistic*). The idea behind this feature is that for most normal flows TTL changes are not random, whereas for the covert channel they are random because the covert bits are uniformly random distributed.

Finally, we used the number of TTL changes divided by the number of packet pairs as another feature (referred to as *NumberChanges*).

7.2.5 Results

Figure 7.4 plots the average flow-based and bit-based F-measure over the encoding fraction for the MED modulation scheme for both traces using all features. The left graph shows the detection accuracy with sparse encoding as described in Section 4.2.2. The right graph shows the detection accuracy with modified sparse encoding described in Section 7.2.2, which we use throughout the rest of this section.

For Twente the flow-based accuracy is over 95% for high encoding fractions, but it decreases to around 83–84% for low fractions. With sparse encoding the accuracy increases again for very low encoding fractions, but with modified sparse encoding it stays low. For Bell the per-flow classification accuracy is over 95% for high encoding fractions. It gradually decreases, but stays over 90% even for low encoding fractions.

The bit-based accuracy is significantly higher for both traces. For Bell it is over 95% even for low encoding fractions, but for Twente it decreases to around 90%. Flows with more covert bits are easier to detect, which is somewhat expected. In general the overall

Table 7.1: Feature sets used for analysing the impact of different features on detection accuracy and classifier complexity

Set	Features
1	ChangePosEntropy, ChangePosMedian, Entropy, EntropyRate, RunsStatistic, NumberChanges
2	ChangePosEntropy, ChangePosMedian, Entropy, EntropyRate,
3	ChangePosEntropy, ChangePosMedian

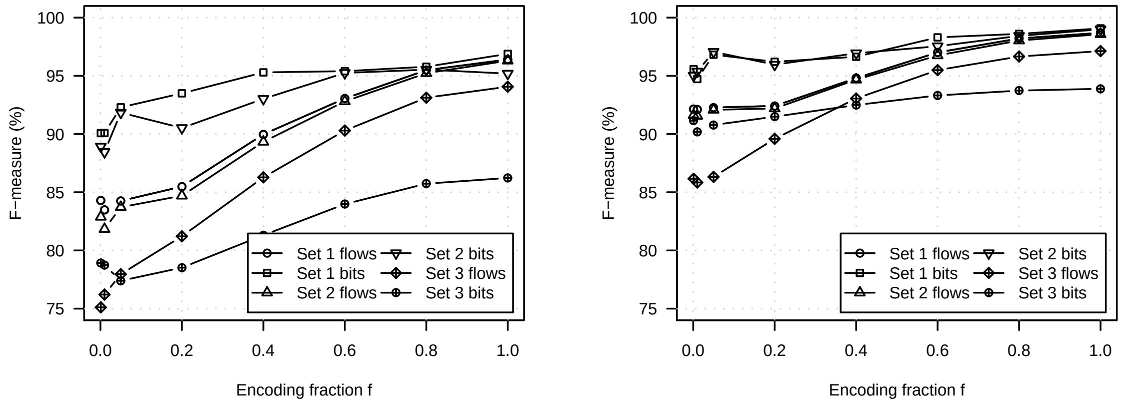


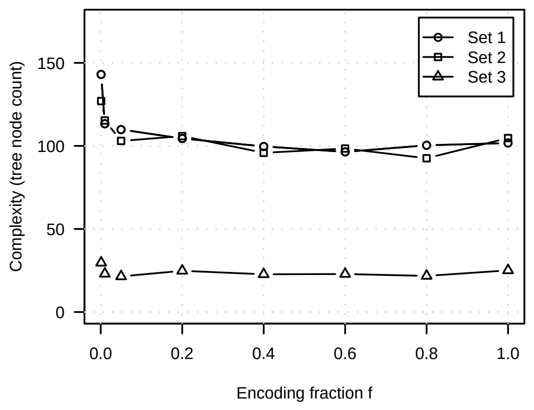
Figure 7.5: Detection accuracy for the MED modulation scheme depending on the feature set for Twente (left) and Bell (right)

accuracy is higher for Bell. Although there are more normal flows with TTL changes in the Bell trace the classifier is able to better separate them from flows with covert channels. The detection accuracy for the DED modulation scheme is similar, although for low encoding fractions it is 1–2% lower (see Appendix B.12).

As mentioned previously, the accuracy of C4.5 is not much affected by irrelevant features. However, the larger the number of features used the more complex the classifier becomes. We now analyse the impact of different feature sets on the accuracy and complexity of the classifier. Table 7.1 lists the different feature sets. Set 1 is the complete set of features. Set 2 and Set 3 are reduced feature sets. We successively removed features that are less relevant because they are less prominent at the top levels of the tree generated with the full feature set.

Figure 7.5 shows the average F-measure for the different feature sets for both traces for the MED modulation scheme. For Set 2 there is only a small accuracy reduction compared to the full set. For Set 3 the decrease is substantial for Twente, but less dramatic for Bell. The results for the DED modulation scheme are similar (see Appendix B.12).

Figure 7.6 shows the complexity of the classifier for the different feature sets for the MED modulation scheme. For Set 2 the complexity is almost the same as for the full feature set. However, for Set 3 it is dramatically reduced. The complexity for the Bell trace is slightly larger because that trace contains more flows with normal TTL changes.



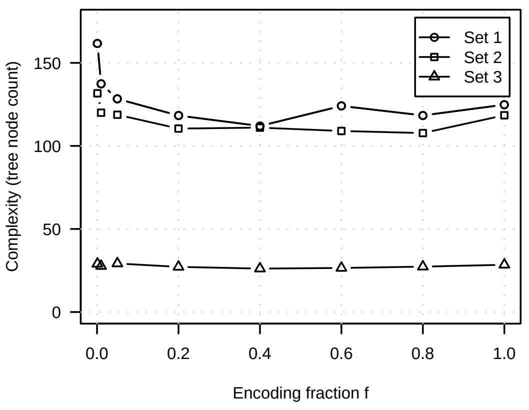


Figure 7.6: Complexity of the classifier for the MED modulation scheme depending on the feature sets for Twente (left) and Bell (right)

For both traces the complexity is largest for very small encoding fractions. Again, the results for DED are similar (see Appendix B.12).

Our results show that for both modulation schemes the TTL covert channel is detected with over 95% accuracy if a large fraction of overt packets is used. Even if only a small fraction is used the accuracy only decreases to about 85–90%, because the TTL covert channel does not mimic normal TTL variation very well.

The complexity of the classifier is almost constant for different encoding fractions, but does increase for very small fractions. It is larger than the complexity of the classifiers for the other investigated covert channels. The complexity can be reduced at the cost of reducing the accuracy by reducing the number of features.

7.3 Elimination of TTL channels

We assume that the warden is located at the perimeter of a protected network, co-located with firewalls or intrusion detection systems. In principle it is not very difficult to eliminate the TTL channel, if the warden is able to manipulate the TTL field of packets. The warden needs to set the TTLs of all packets of a flow to the same value, i.e. ‘clamp’ the TTL value.

Ideally, the warden should set the TTLs to the smallest TTL value of the flow observed thus far, thereby avoiding increasing any TTLs. However, this approach requires that the warden maintains per-flow state (lowest TTL), which may be problematic if CPU or memory resources are limited. Furthermore, this technique opens the door for a new DoS attack if an attacker is able to spoof valid packets of an existing flow. Then the attacker can send packets with arbitrary low TTL forcing all following packets to be dropped before they reach the destination.

An alternative solution is to set each TTL to an arbitrary value smaller than the original TTL but large enough so that packets are not dropped before reaching the destination. Modern operating systems use initial TTL values of at least 64 [167, 168], and the maximum number of hops in the Internet is typically less than 32 [167]. For packets coming into the protected network the warden can reduce the original TTL value to a small value equal or slightly larger than the known maximum number of hops inside the network. For outgoing packets the warden can reduce the TTL to 32. This approach does not require per-flow state and does not create opportunities for new DoS attacks.

It is quite unlikely that the original TTL values are smaller than the clamp value, unless a covert channel is used. However, should this happen the warden would have to increase TTL values. To avoid this, the warden should not change the TTL values, but instead monitor or block the suspicious flow. A warden that is not able to manipulate all packets of a flow cannot completely eliminate the channel, but would probably reduce its capacity because of the added noise.

7.4 Detection of packet-timing channels

Now we analyse the detection accuracy for covert channels in inter-packet times (also referred to as IPGs). Previous work on the detection of IPG timing channels only used simple tests based on single features [107]. We evaluate classifiers that use multiple features to differentiate between the classes. First, we describe the datasets and features used. Then we present the results of our analysis.

7.4.1 Datasets and methodology

As normal traffic we used the datasets introduced in Section 4.1. The covert traffic was created based on these datasets using sparse encoding or sub-band encoding described in Chapter 4. For each normal flow we generated one corresponding covert channel based on the same IPG distribution. The covert data was uniformly random distributed, as it would be if covert sender and receiver used encryption.

Each dataset had the same number of covert and normal flows to avoid bias of the classifier towards a larger class (see Table 7.2). As in Section 4.1 we used the first 5 000 IPGs of each flow. Since each instance of the covert class has roughly the same number of covert bits, bit-based accuracy is equivalent to flow-based accuracy. For sparse encoding we evaluate the detection accuracy depending on the encoding fraction f. For sub-band encoding we evaluate the accuracy based on the size of the least significant part l.

The classification accuracy and classifier complexity vary depending on the random covert data and the random number series T and R used for encoding (see Section 4.2.2).

Table 7.2: Datasets for evaluating the detection accuracy and classifier complexity for inter-packet gap timing channels

Dataset	Normal flows	Covert flows
Q3 client-to-server	106	106
Q3 server-to-client	224	224
Skype	44	44
Twente UDP	111	111
Twente TCP	220	220

Furthermore, some variation is introduced since during the cross-validation instances are randomly selected for either the training or the testing set. Therefore, we performed cross-validation 10 times and report average performance metrics.

7.4.2 Features

Let $X = [X_1, \ldots, X_n]$ be the series of IPGs of a flow with values $x_1, \ldots, x_n$. For each X we computed the Entropy and the EntropyRate. To determine the number of bins Q for the entropy rate we performed several initial tests and found $Q = 5$ provided the best results.

We computed another feature based on the two-sample Kolmogorov-Smirnov (KS) test. This test tests the hypothesis that two samples were drawn from the same distribution. A low KS test statistic means that the distributions are similar whereas a high KS test statistic means the distributions are different. The KS test is distribution-free, meaning it is applicable to a variety of types of data with different distributions.

The KS test computes the test statistic for two distributions. What we need however is a feature that reflects how different a distribution is from the set of distributions characterising the normal traffic. Therefore, we computed the set of KS test statistics between the data instance (covert or normal) and all instances of normal traffic (excluding a test of a normal instance with itself) using the R function `ks.test()` [239]. We used the mean of the set of KS statistics as feature (referred to as *MeanKS*).

7.4.3 Results

First, we analyse the detection accuracy and classifier complexity for sparse encoding. Figure 7.7(left) shows the F-measure averaged over both classes for the different datasets depending on f. The results for $f = 1.0$ show that the covert channel in [104, 105] is detected with high accuracy. The TCP and Q3 server-to-client traffic has the least auto-correlations and consequently the detection accuracy is smallest. However, it is still in the range of 90–95%. Averaged over all datasets the detection accuracy is over 95%.

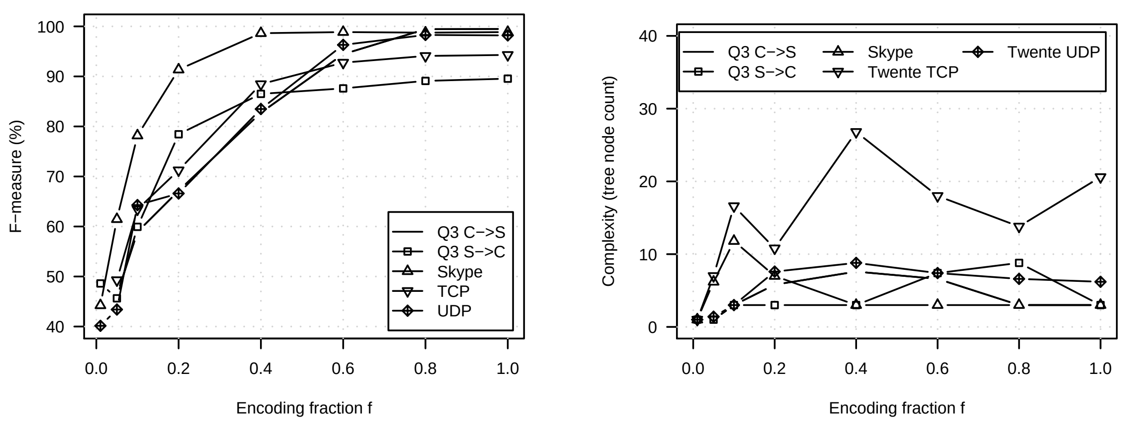


Figure 7.7: Detection accuracy (left) and classifier complexity (right) for the different datasets using sparse encoding

When f is reduced the detection accuracy is also reduced, especially for $f < 0.4$. For smaller f the detection accuracy is similar for the different datasets, except for Skype traffic it is higher because of the higher auto-correlations.

Figure 7.7(right) shows that the classifier complexity is small. For the UDP-based traffic it is generally less than ten nodes. For TCP it is consistently higher, because of the wider variety of traffic characteristics. Since the covert channel mimics the shape of the distributions of the IPGs very well, the Entropy metric is not useful. The EntropyRate metric measures the complexity of the series of IPGs and differentiates well between the covert channel with iid IPGs and normal traffic with auto-correlated IPGs. The MeanKS metric is occasionally used at the bottom of the tree.

Now we analyse the detection accuracy and classifier complexity for sub-band encoding. Figure 7.7(left) shows the F-measure averaged over both classes for the different datasets depending on l. The detection accuracy is very low for small l. It increases with increasing l, but even for larger values it only reaches 80–90%. For $l = 5$ ms the detection accuracy is only up to 70–80%.

Figure 7.7(right) shows that the classifier complexity is slightly higher than for sparse encoding. However, in most cases the classifier still only has up to 20 nodes. Again, the classifier relies mainly on EntropyRate, but MeanKS is also used occasionally at the bottom of the tree.

The timing channel proposed in [104, 105] is difficult to detect when IPGs are iid, which is the case when packet send times directly depend on human actions (e.g. Telnet). However, it is easy to detect with over 95% accuracy if there are correlations in IPGs, which is the case for several network applications (see Section 4.1). Our new encoding schemes are harder to detect with accuracies of only up to 70–80%. The stealth is im-

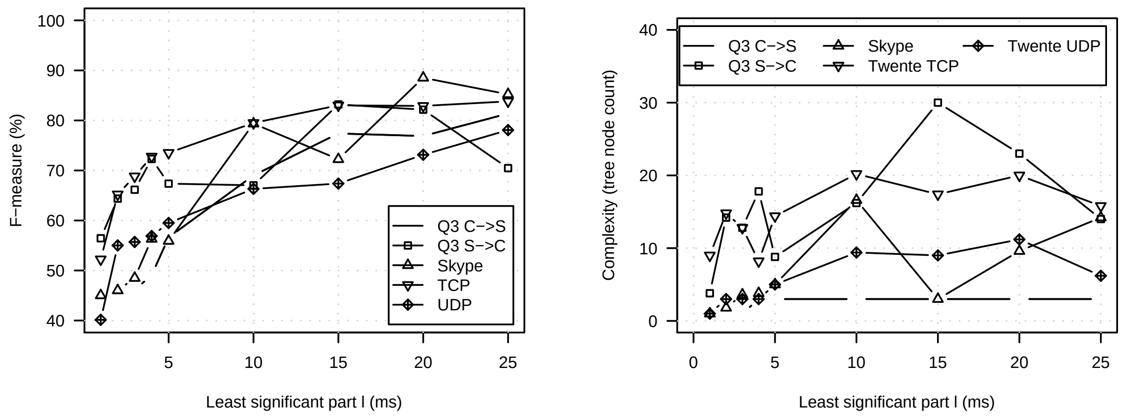


Figure 7.8: Detection accuracy (left) and classifier complexity (right) for the different datasets using sub-band encoding

proved by reducing the channel capacity (sparse encoding) or the robustness (sub-band encoding). We think these are good trade-offs as stealth is paramount for covert channels.

The buffering delay introduced could reveal (semi-)passive covert channels, but it is difficult to detect. The warden would have to know the usual network delay and the delay the covert channel experiences. Measuring the usual delay is difficult, since the warden typically has no access to end hosts, and covert sender and receiver could tamper with active measurements. Measuring the delay of the covert channel is also not easy. For TCP traffic it could be estimated (e.g. [240]), but for UDP-based traffic there is no general solution. A delay-based detection of the channel may be possible, but it appears to be much harder than an analysis of IPGs, the sequence of messages or the packet payload.

Another possible approach for detecting (semi-)passive timing channels is to let the overt sender embed the actual packet send times in the packets. The timestamps must be secured by either encrypting or protecting them with digital signatures. Then the channel can be detected if the covert sender's manipulations exceed typical noise. However, this approach would require modifications of existing protocols and network stacks.

7.5 Elimination of packet-timing channels

In Section 4.5 we showed that timing jitter greatly reduces the channel capacity and effectively eliminates the channel. For Pareto-distributed jitter with $\sigma \geq 2$ ms, the capacity of the channel reduces to one bit per second or less (see Figures 4.14 and 4.16).

If the warden does not know which traffic flows carry covert channels, she may 'blindly' introduce artificial jitter into the IPGs of all flows in order to limit the capacity. This practically eliminates the covert channel possibly present in a few flows. However, it also unnecessarily introduces network jitter into a large number of legitimate flows.

Here we examine how the deliberately introduced inter-packet timing jitter affects the performance of the applications that generate the overt traffic.

We used the same testbed as described in Section 4.5. This time we used Netem [189] to introduce timing jitter into the overt channel without any covert channel present. We emulated Pareto-distributed network jitter with a mean of 25 ms and standard deviations of 0.3, 0.5, 1, 2, 3, and 5 ms[5]. The Linux kernel's tick frequency was set to 10 kHz, so the emulated delays were accurate to about $\pm 100\,\mu$s.

As before, we used scp file transfer, interactive SSH and Q3 game traffic as overt traffic (see Section 4.5). As performance indicators we measured the data throughput (scp) and the jitter (SSH and Q3). We performed each experiment three times and report the average throughput and jitter. We also performed preliminary tests with a human operator (SSH) or player (Q3) to gauge the effects of the network jitter on the usability of the two applications. We leave a comprehensive usability study for future work.

The latency of the actual application traffic was measured passively using the Synthetic Packet Pairs (SPP) technique [241]. During the experiments we recorded the traffic at both hosts, and then computed latency by halving the RTTs estimated by SPP after the fact (since emulated delay was symmetric).

Figure 7.9 shows the average TCP throughput, normalised based on the maximum throughput with zero network jitter[6], depending on the increasing network jitter. In the first experiment scp limited the bandwidth to 2 Mbit/s as before, but in the other two experiments there was no limit. In the third experiment we used a smaller mean network delay of 5 ms in each direction.

When scp limits the bandwidth the throughput does not decrease with increasing jitter because scp 'compensates' any throughput fluctuations of the TCP protocol. Otherwise, the throughput reduces down to 80% (25 ms RTT) or 55% (5 ms RTT).

With $\sigma = 2$ ms, where the covert channel is practically eliminated, the throughput is still approximately 93% for the same network delay used to evaluate the channel in Section 4.5. We think such a relatively small reduction is acceptable in order to limit the capacity to the common limit of one bit per second [19]. With smaller network delays the throughput reduction is much more substantial, but it is questionable if a covert channel could work with such high packet rates given the inaccuracies inherent in timing packets. In general the throughput reduction depends on the network delay and jitter, the artificial network jitter, and the TCP protocol variant and configuration[7].

Figure 7.10 shows the IPDV [201] distributions for SSH and Q3 traffic. As expected, the distributions look very similar for both applications. Even for $\sigma = 5$ ms the IPDV

[5]The jitter was introduced without reordering of packets.

[6]Maximum throughputs without network jitter were approximately 2 Mbit/s, 11.5 Mbit/s and 40 Mbit/s.

[7]We used TCP Reno with the default configuration of Linux 2.6.20.

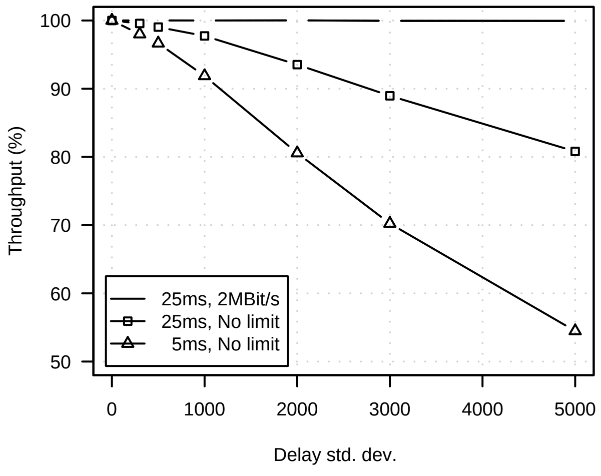


Figure 7.9: Average TCP throughput depending on the standard deviation of the emulated network jitter (artificial noise)

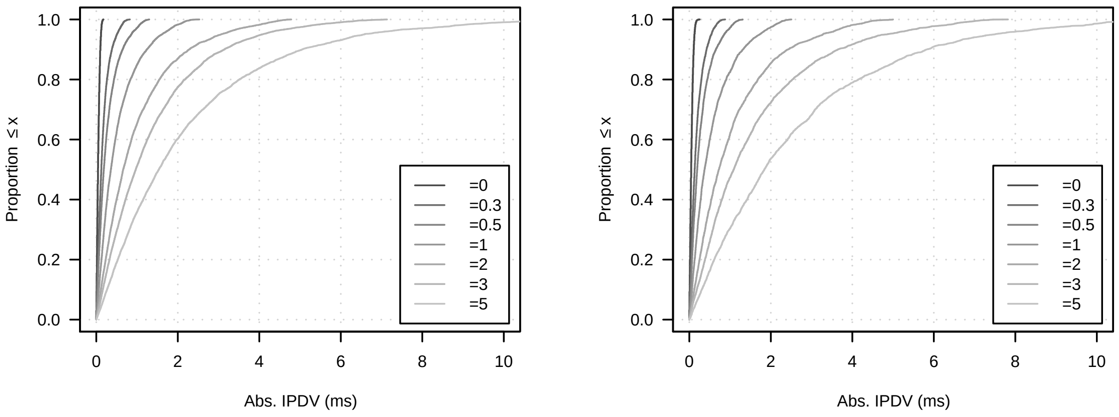


Figure 7.10: Absolute IPDV for Q3 (left) and SSH (right) traffic depending on the standard deviation of the emulated network jitter (artificial noise)

is mostly contained within ±10 ms. No study exists on the impact of jitter on Q3, but previous work for other FPS games showed that jitter of this magnitude has only little impact on the quality of games and does not affect the performance of players [242]. Since Q3 arguably has stricter latency and jitter requirements than SSH, the effect for SSH should be negligible. Preliminary experiments with a human user did not show any adverse effects on the usability. The introduced jitter was not noticeable.

Our results show that the covert channel is practically eliminated with only small performance degradations of the applications generating the overt traffic. However, channels in overt traffic with wider IPG distributions are more robust against network jitter. Since the warden cannot introduce high jitter into high-rate TCP flows or flows of highly-interactive applications without affecting their performance, she must select the appropriate noise level based on a flow's application type and IPG distribution.

Flows with wider IPG distributions have comparatively low packet rates, however covert sender and receiver could use multiple such flows. This means the warden must not only analyse the traffic on a per-flow level, but also should look at patterns of multiple flows over time. For example, many simultaneously active SSH terminal sessions between two hosts could be considered as suspicious.

It is possible that other jitter distributions are more effective in eliminating the covert channel with fewer 'side-effects'. We leave a more in-depth study as future work.

7.6 Detection of multiplayer-games channels

First we discuss varies ways Wendy could detect FPSCC. Then we describe the datasets and features used in our analysis. Finally, we evaluate the detection accuracy and classifier complexity.

7.6.1 Detection strategies

We assume that Wendy is usually one or multiple client(s) connected to the same server as Alice and Bob and logs all received player movement updates (although covert data would only be received when Wendy is visible to Alice's or Bob's player). Since the number of existing servers is typically very large, Wendy might monitor a smaller subset of suspicious servers. Wendy could also be located on the server and log all player movements, but it seems practically impossible to deploy her on thousands of servers. However, Wendy could set up some 'honeypot servers' to attract covert channel users.

FPSCC cannot be detected because of the small latency it adds. Our passive middleman implementation introduces only 1–2 ms additional latency. Given that in-game client-side reporting of 'ping' is not very accurate, FPSCC is unlikely to be discovered by players monitoring their ping. Furthermore, we assume that Alice and Bob always play with FPSCC enabled and hence Wendy does not know their regular ping statistics.

Slight angle movements of another player character are basically invisible to human players. The game server already introduces errors of up to one degree by only distributing the integer values of angles (see Section 5.2.7). If Alice is a middleman, there is a greater chance that the player whose character's movements are being modulated might notice abnormalities, since the server transmits view angles to the player as full floating point numbers. However, Alice could 'clean' the view angles being sent back to the client from the server.

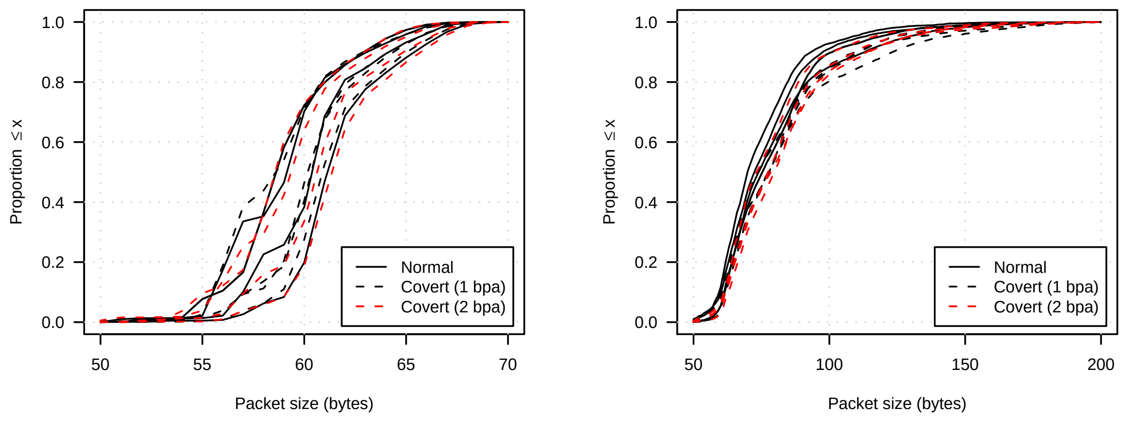


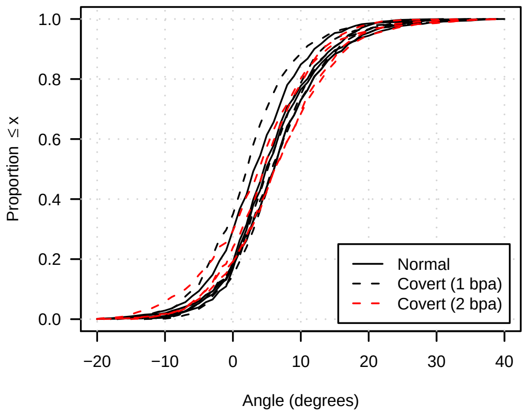
Figure 7.11: Packet length distributions in client-to-server (left) and server-to-client (right) direction for normal Q3 traffic and RFPSCC with one bit and two bits encoded per angle change

FPSCC may affect the aim of players and hence their performance. However, there are a variety of other factors affecting their performance, such as network jitter, opponents, playing strategy or fatigue. Furthermore, again we assume that Alice and Bob always play with FPSCC enabled and hence Wendy does not know their usual performance. In our experiments with nine human players described below, none of the players noticed the view angle fluctuations RFPSCC introduced, not even in their own uncleaned view angles. While none of the players attributed changes in their own performance to RFPSCC, two players noticed that their opponent performed better without RFPSCC.

Nevertheless, even if invisible to players, FPSCC affects the view angles. Because Q3 messages are delta encoded and Huffman-compressed, packet sizes could be affected as well. Figure 7.11 compares the packet sizes of RFPSCC versus normal traffic for 1 bpa and 2 bpa. Each graph shows four datasets for normal traffic, RFPSCC with 1 bpa and RFPSCC with 2 bpa, each randomly selected from the overall data.

For the packet size distributions the differences between different players are larger than the differences between RFPSCC and normal traffic. Q3 is known to exhibit wide variation in packet sizes versus the number of players and the map played [243]. Without knowing the normal distributions for each combination of map and number of players, it is very difficult for Wendy to detect RFPSCC. In practice it is even harder because the number of players changes over time. However, in our analysis we have the ideal case where Wendy only compares distributions for the same map and same number of players.

Figure 7.12 shows the angle distributions of RFPSCC and normal traffic. Each graph has four datasets for normal traffic, RFPSCC with 1 bpa and RFPSCC with 2 bpa, randomly selected from the overall data. For pitch and yaw the differences between different players are larger than the differences between RFPSCC and normal players. If Wendy



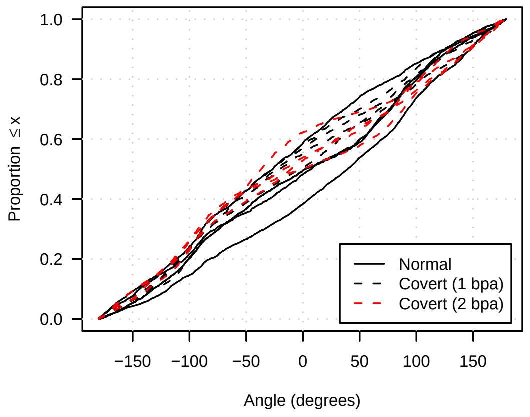


Figure 7.12: Angle distributions for pitch (left) and yaw (right) for normal Q3 traffic and RFPSCC with one bit and two bits encoded per angle change

does not know the usual angle distributions of a particular player, it is difficult for her to detect RFPSCC, especially as the number of players and the map may also impact on the distributions. However, knowing RFPSCC's modulation technique Wendy could also compare modulo delta angles (see Section 7.6.3).

7.6.2 Datasets and features

We collected data from a number of games with nine different human players. Each player played three five-minute games, one without RFPSCC, one with RFPSCC with 1 bpa and one with RFPSCC with 2 bpa. To establish equal conditions every human player played against a bot. Our test games were carried out with an RTT of 25 ms and zero packet loss. The RTT does not have an impact on the detection accuracy and packet loss would only make it harder to detect RFPSCC because of the incomplete data.

In total we collected data for 15 sets of games since some players played more than once, and we also included the bot as another player. To increase the number of data instances we divided the existing data into subsets of 500 consecutive samples each and use each subset as separate instance. We chose 500 samples as size of the subsets because classification accuracy decreases for smaller sets as shown in Figure 7.16 and initial tests indicated that there is very little gain for larger sets.

To ensure that in each training dataset RFPSCC and normal classes have the same number of instances, we always reduced the larger class to the size of the smaller class using random sampling. For packet sizes and yaw there are 60 instances per class, but for pitch there are only 44 instances per class. Since our datasets are small there is considerable variance in the performance metrics depending on how instances are selected during

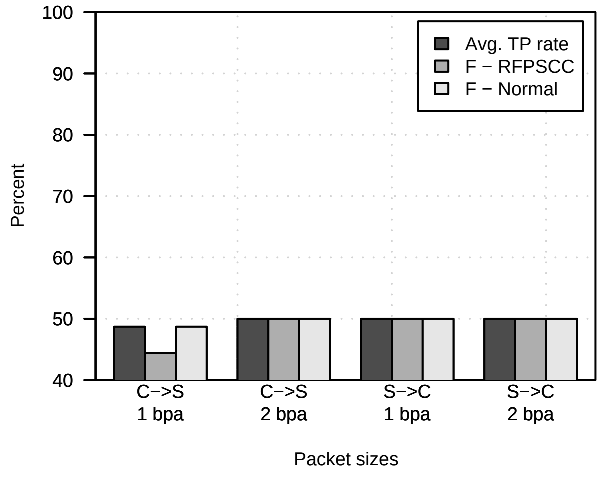


Figure 7.13: Accuracy of detecting RFPSCC based on packet length distributions in client-to-server and server-to-client direction for one bit and two bits encoded per angle change

the cross-validation. Therefore, we performed cross-validation ten times and report the average performance metrics.

Let $X = [X_1, \ldots, X_n]$ be the series of packet lengths or view angles with values $x_1, \ldots, x_n$. The first two features we computed are the Entropy and the EntropyRate of X. To determine the number of bins Q for the entropy rate we performed initial tests. For packet lengths and view angles we found $Q = 5$ provided the best results. For modulo delta angles Q was set equal to the modulo. We also computed the MeanKS feature introduced in Section 7.4.

7.6.3 Results

First we investigate if RFPSCC can be detected based on observed packet lengths. Figure 7.13 shows the average TP rate, and the per-class F-measure for packet lengths in client-to-server and server-to-client direction for 1 bpa and 2 bpa encoding. The results show that packet length cannot be used to detect RFPSCC. In most cases C4.5 fails to build a classifier and the average accuracy and F-measure are 50% (equal to random guessing).

Next, we examine the effectiveness of detecting RFPSCC based on angle distributions. Figure 7.14 shows the average TP rate, and the per-class F-measure for pitch and yaw for 1 bpa and 2 bpa.

The results show that RFPSCC in yaw cannot be reliably detected. Since the yaw distribution is wide (-180 to 180 degrees) compared to the small changes of RFPSCC, the classifier cannot distinguish between both. However, the pitch distribution is narrower since pitch varies only between -20 and 60 degrees (see Figure 7.12). Hence relatively the RFPSCC changes are larger and the classifier has higher accuracy. Still the average TP

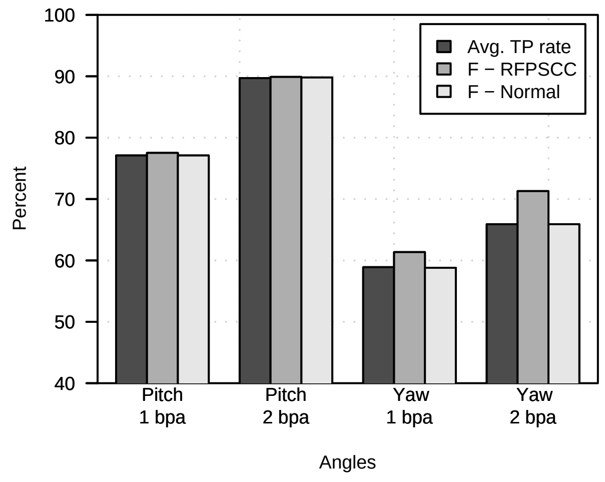


Figure 7.14: Accuracy of detecting RFPSCC based on pitch and yaw angle distributions for one bit and two bits encoded per angle change

rate is less than 90%. As expected, the more bits are encoded per angle change the easier it is to detect RFPSCC. The feature providing the best discrimination is the EntropyRate, but MeanKS and Entropy are also used in case of pitch.

A skilled warden, who is aware of RFPSCC and its encoding, could compare the distribution of modulo delta angles. In practice Wendy would not know the number of bits encoded per angle, but could test against classifiers build for different moduli. Figure 7.15 shows the average TP rate, and the per-class F-measure for pitch and yaw for 1 bpa and 2 bpa.

For delta modulo pitch 1 bpa the classifier achieves an average TP rate of 98% or higher. For delta modulo yaw 1 bpa the classifier achieves average TP rates of approximately 96%. The accuracy decreases slightly with increasing number of bits encoded per angle. No single feature stands out, but EntropyRate and Entropy are used more frequently and further up in the tree than MeanKS.

We also investigated how quickly RFPSCC can be detected, i.e. how many angle samples are needed. We performed cross-validation with the first 25, 50, 100, 200, 300, 400, 500 samples of the datasets using modulo delta angles. Figure 7.16 shows the average TP and FP rates depending on the number of samples.

The TP rate is quite low and the FP rate is very high for only a small number of samples. At least 400–500 samples are needed to increase the accuracy to acceptable levels. As discussed in Section 5.5.1, the average number of view angle changes for human players was approximately 6 changes per second for pitch and 9 changes per second for yaw. This means Wendy can reliably detect RFPSCC by observing Alice for at least 45–65 seconds, assuming Alice is sending continuously.

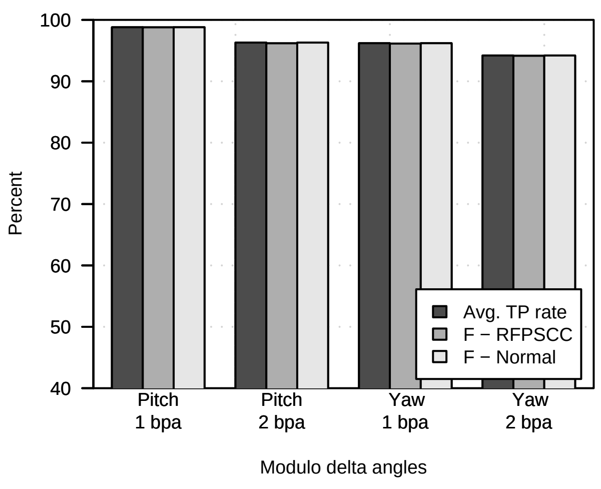


Figure 7.15: Accuracy of detecting RFPSCC based on modulo delta pitch and yaw angle distributions for one bit and two bits encoded per angle change

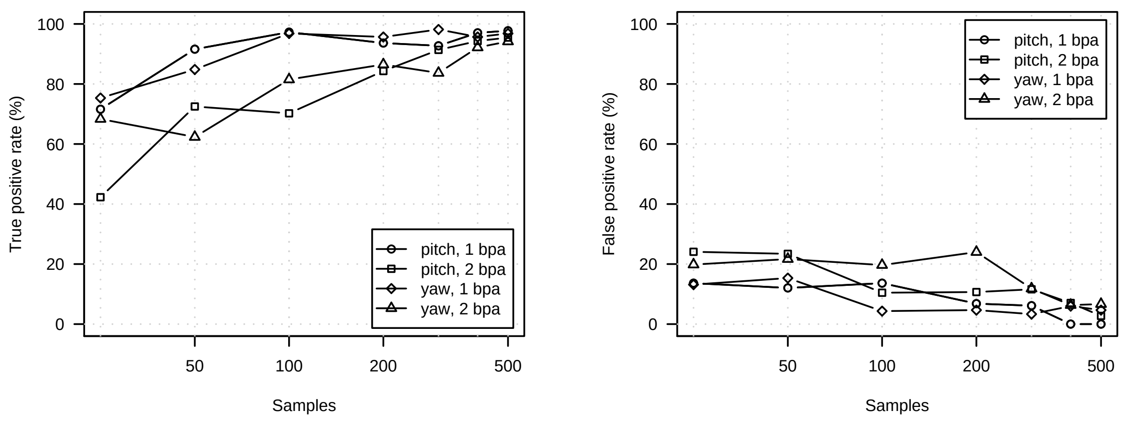


Figure 7.16: True positive rate (left) and false positive rate (right) depending on the number of modulo delta angle samples (x-axis is log scale)

In 45 seconds RFPSCC can transmit approximately 270 bits, given our results in Section 5.5 (assuming two players, 2 bpa encoding, an RTT of 75 ms and a packet loss rate of 0.5%). If Alice sends shorter messages, she may evade detection.

Classification performance can be further improved by considering classification costs. False negatives can be reduced at the cost of increasing false positives or vice versa. A cost matrix is used to specify classification cost and then C4.5 considers the costs during the training phase. For example, the cost matrix:

$$C = \begin{bmatrix} 0 & 10 \\ 1 & 0 \end{bmatrix}, \tag{7.13}$$

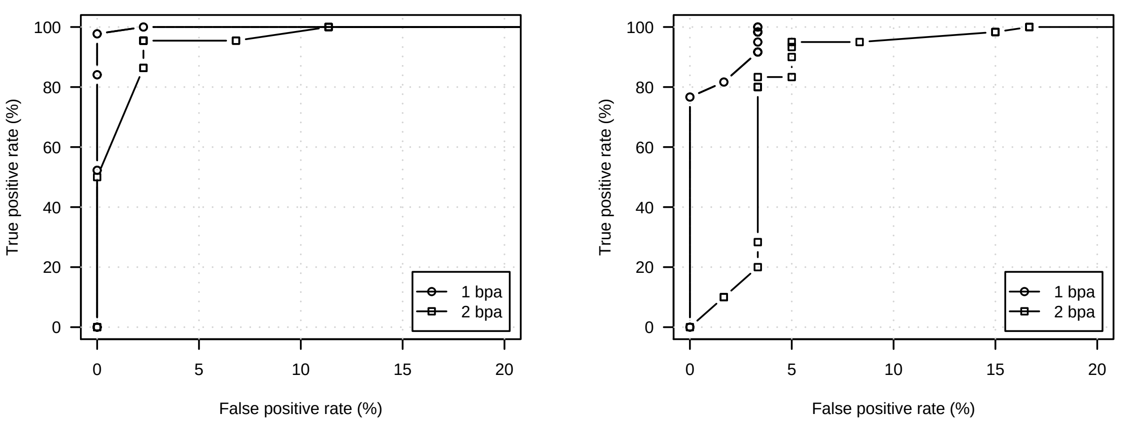


Figure 7.17: ROC curves for detecting RFPSCC based on modulo delta pitch (left) and yaw (right) depending on the number of bits encoded per angle

means it is ten times costlier to classify an instance of RFPSCC as normal than to classify a normal instance as RFPSCC (assuming RFPSCC is the first class).

Figure 7.17 shows the Receiver Operating Characteristics (ROC) curves for pitch and yaw. For pitch with 1 bpa a very high TP rate is achieved even with zero false positives. However, in all other cases a very high TP rate is only reached with false positive rates of at least 3–5%. Another possible way to increase accuracy is to use two separate classifiers for pitch and yaw and combine their results. We leave this as future work.

We showed that while RFPSCC looks very similar to normal game traffic, it is detected with an accuracy of at least 95%. However, usually the number of false positives is still in the order of 3–5%. It appears likely that the detection accuracy is smaller in practice when the set of different players is very large. Packet length and angle distributions cannot be used to detect RFPSCC reliably. Only using the modulo of the angle differences provides accurate detection.

Since RFPSCC must impose some structure on the bits sent to provide bit synchronisation, it cannot exactly mimic normal behaviour. This is similar to other covert channels that suffer from bit insertions or deletions. RFPSCC's stealth could be improved in future work by better disguising these structures.

Note that FPSCC (but not RFPSCC) could be used for unidirectional transmission. Then, even if Wendy detects the channel and identifies Alice, Bob is still partly protected. If Alice is using broadcast mode, Wendy cannot know which of the client(s) acting as Bob(s). In unicast mode, Alice can interfere with Wendy identifying Bob by sending 'garbage' with similar characteristics to FPSCC, when Bob is not in range.

7.7 Elimination of multiplayer-games channels

FPSCC cannot be eliminated without eliminating the game traffic, because player movement is intrinsic to the game. Additional noise could be introduced to limit the channel capacity, such as the game server introducing minute, random fluctuations in every player's view angles. However, such noise must have no visible consequences for the players.

But Alice could always counter this noise by increasing the 'sending power'. If Alice is also the overt sender then she is actually playing the game, in which case she can use larger view angle fluctuations possibly visible on her screen, while Wendy has to stay below a threshold where the changes become visible. Wendy could only effectively jam FPSCC by first detecting Alice's use of the channel and then inserting large noise specifically into her angle changes or disrupting her traffic (e.g. disconnect her).

7.8 Measures against temperature-based channels

It is very difficult to completely eliminate temperature-based covert channels. However, a number of measures can be employed to reduce their capacity.

A seemingly obvious way of eliminating the channel is to prevent Bob's sampling of the clock by removing all timestamps from network protocols. However, this has a negative effect on the performance or functionality of protocols. For example, the TCP timestamp extension is needed for improving the performance of TCP and the HTTP timestamp is needed for HTTP caching. Furthermore, many low-level operating system events are triggered on timer interrupts and could be remotely detected and used instead of explicit timestamps [221].

A clock crystal that is not affected by temperature changes eliminates the channel. However, temperature-compensated crystals might not have adequate accuracy [244]. Oven-compensated crystals have good accuracy, but are very expensive and power hungry [244]. Thus it seems unlikely that accurately compensated crystals would ever be deployed widely.

The opportunity for Alice to induce CPU load cannot be completely eliminated. However remote load inducement could be limited if the network traffic is throttled before it reaches the intermediate host or in the worst case on the intermediate host itself before it reaches an application. If Alice is located on the intermediate host a similar measure would be to limit the amount of CPU time a process or user can use.

Another countermeasure is to increase the channel noise by randomly varying CPU load on the intermediate host or in the extreme case by continuously running the CPU at full load. However, this strategy is obviously very inefficient. Furthermore, care must be taken in its implementation because the temperature does not only depend on the CPU load but also on the specific mix of instructions executed and hence different types of tasks can have different temperature effects [244].

The channel should be relatively easy to detect if Alice tries to maximise the transmission rate by generating large CPU load changes, and thus sends large amounts of traffic. However, Alice and Bob could always vary their traffic or CPU load patterns trying to evade detection, although this would most likely also reduce the transmission rate. Then detection of the channel may not be straightforward. A warden would have to look for abnormal traffic or CPU load patterns indicating either Alice or Bob. Hence the detection accuracy depends on the regular patterns at the intermediate host.

7.9 Conclusions

We proposed a number of channel-specific and effective techniques for eliminating the different covert channels.

The TTL channel is not difficult to eliminate. Even in rare cases where it cannot be completely eliminated at least its capacity is reduced. The inter-packet gap timing channel cannot be completely eliminated, but we showed for a small number of example applications that the channel can be eliminated with only minor side-effects on the performance of the overt traffic.

It is hard to eliminate the temperature-based covert channel completely. However, given that the channel's capacity is very low, it only requires handling in certain scenarios, such as anonymisation networks. Furthermore, we discussed some measures that can be applied to further reduce the channel's capacity.

Elimination of covert channels is not always possible, as the game traffic channel demonstrates. Detection is another important countermeasure. Even if the channel could be eliminated, the warden may choose not to do so because successful detection usually allows identifying the covert sender and receiver. Furthermore, demonstrated detection acts as deterrence to possible users of the channel.

We investigated how effectively ML techniques can detect different covert channels. We proposed a number of characteristics to separate covert channels from normal traffic. While some of the features proposed were generic and were used for different covert channels, some were tailored to specific channels. We used the C4.5 decision tree algorithm because it showed good performance in previous work [235].

Even low-amplitude TTL covert channels are easy to detect if encoded in a high fraction of overt packets, because normal TTL variation is infrequent. For high encoding fractions the TTL covert channel is detected with over 95% accuracy. With smaller encoding fractions the accuracy is reduced. However, it is still in the order of 85–90%, because the channel does not mimic normal TTL variation very well.

The inter-packet gap timing channel proposed in [104, 105] is easy to detect with over 95% accuracy if the IPGs of normal traffic are auto-correlated. Our new encoding schemes are harder to detect with accuracies of only up to 70–80%, since they mimic the shape and auto-correlation of normal IPGs well. While detection of the covert channel embedded in game traffic is non-trivial, it can be detected with accuracies of over 95% if the warden knows its modulation technique.

Our work demonstrates that some covert channels can be eliminated but are hard to detect (new IPG timing channels), while others can be detected but are hard to eliminate (game traffic covert channels). This means in order to handle different types of covert channels future security systems must implement both countermeasures.

7.9.1 Future work

Future research could develop new more effective features to further improve the detection accuracy and study the performance of other ML algorithms. We think that developing better features is more promising than evaluating other ML techniques, since the better ML algorithms usually provide similar classification accuracy [235].

The elimination of IPG timing channels needs further research. Our results are promising, but we have only investigated one specific technique. Future work should evaluate other noise distributions or noise-introducing techniques that limit the channel capacity with only minimal side-effects. As discussed earlier, adaptive techniques are needed that introduce noise at different levels based on the characteristics of the overt traffic. Usability trials may be needed to quantify the effects of eliminating covert channels on interactive applications, such as online games.

We described how to eliminate TTL channels, but an implementation of the proposed method does not exist yet. Another avenue of future research is the analysis of countermeasures against the temperature-based channel. This channel may become more attractive to users in the future if other higher-capacity channels are handled by the next generation of security systems.

Our software framework, CCHEF, could be modified in order to use it not only for creating covert channels, but also for detecting and eliminating them. This would allow testing and evaluating the countermeasures in real networks.

Chapter 8
Conclusions and Future Work

Historically, covert channels were identified as instruments to violate security policies by leaking information from a high-security process to a low-security process on monolithic Multi Level Secure (MLS) systems, typically military systems. With the emergence of low-cost, highly distributed and networked computer systems, the focus of research has expanded to include network-protocol covert channels between different hosts.

The possible applications for covert channels have extended far beyond the MLS scenario. For example, government agencies or criminals may use covert channels for communication, hackers or spies may use them for data ex-filtration or controlling hacked systems, or the general population may use them for circumventing censorship or company security policies.

As discussed in Chapter 2 many existing covert channels are simple noise-free channels. They are easy to implement, but they are also easy to eliminate and detect. Very little variation of normal behaviour of traffic sources and the absence of noise make these channels stand out. Furthermore, they can be removed easily through protocol normalisation. There are a few more complex channels, which are harder to detect and eliminate, but suffer from noise. Their performance has not been studied much in previous work. Furthermore, previous research never compared different types of channels.

In this thesis we analysed and compared the performance of several selected noisy covert channels. We compared the channels based on experiments in real networks and information-theoretic concepts. Not all of the chosen channels are entirely new, but we usually developed novel improved encoding schemes. We also developed protocols for reliable data transport and analysed their throughput. Finally, we developed countermeasures against the channels and evaluated their performance.

We demonstrated that the potential of the Internet to support sophisticated covert channels is considerably greater than suggested by the simple channels developed so far. We developed noisy storage and timing channels that are not trivial to detect or eliminate, yet provide sufficient capacity for covert communications. We also showed that multiplayer games provide hitherto unexplored possibilities for covert channels and temperature-based covert channels are possible. Taken together this means that security policies and technologies need to become significantly more aware of covert channels.

Building covert channels and observing their behaviour provides an excellent means to analyse them. However, development can be time consuming. Instead of building each channel separately we developed a software framework for implementing covert channels described in Appendix A. Our framework is modular and can be extended easily. It embeds covert channels in existing overt traffic and provides an extensible mechanism for the selection of overt packets used for the covert channel. The design of the framework proved successful as it was barely modified through the duration of the thesis.

In Chapter 3 we analysed a noisy storage channel in the Time-to-live (TTL) field. We analysed the characteristics of the noise based on traffic traces. Normal TTL changes only occurred in less than 1% of packet pairs, but in 2–6% of the flows. The noise is low enough to provide capacities of more than 0.9 bits per overt packet, but large enough so that the channel is not as obvious as simple noise-free storage channels.

We developed new encoding schemes that can be used for passive covert channels, are stealthier and provide up to 5% higher capacities than existing schemes. We proposed a channel model that allows computing the channel capacity based on TTL noise and overt packet loss and reordering. The capacity of the channel ranges from a few tens of bits per second up to a few hundreds of bits per second for single overt traffic flows, and up to thousands of bits per second when encoding in large traffic aggregates.

We showed in Chapter 7 that the TTL channel is detected with over 95% accuracy if a large fraction of the overt traffic is used. Even for very small fractions the detection accuracy is still 85–90%, because the channel does not mimic normal TTL variation very well. Since the TTL field is manipulated by routers, elimination is not as trivial as for simple storage channels, but we proposed an efficient method to eliminate the channel.

Recently noisy timing covert channels were proposed that encode data in inter-packet times. In Chapter 7 we showed that with the existing encoding scheme the channel is still easy to detect with over 95% accuracy if normal inter-packet times are auto-correlated. In Chapter 4 we showed that for several applications this is often the case. We developed new improved encoding schemes that have reduced capacity or robustness but are harder to detect with accuracies of less than 70–80% even if inter-packet times are correlated.

If network jitter is relatively small, the capacity is 70–80% of the TTL channel's capacity. The capacity is still at least several tens of bits per second for single overt traffic flows as shown in Chapter 4. However, it can be drastically reduced to less than one bit per second by introducing artificial network jitter. We showed in Chapter 7 that the channel is practically eliminated with only small negative impact on the performance of the applications generating the overt traffic.

In Chapter 3 we proposed new reliable transport schemes for noisy covert channels, including noise from overt packet loss and reordering. In Chapter 3 and 4 we showed that these schemes achieve throughputs of 50% of the capacity for TTL channels and 30–40% of the capacity for inter-packet gap timing channels.

The channels analysed in Chapters 3 and 4 are direct channels, where covert data is transmitted directly from covert sender to receiver. We also investigated indirect channels where an intermediate host acts as proxy, which increases the security.

In Chapter 5 we demonstrated the potential for novel covert channels based on emerging applications by developing and analysing an indirect noisy storage channel using multiplayer game traffic as cover. The capacity is only up to 10–20 bits per second, but this is still sufficient for covert messaging or chatting. Since the channel is noisy we developed a tailored mechanism for reliable data transport, which achieves over 77% of the capacity. The channel could also be used in immersive virtual worlds that are similar to games. Although interesting in itself our work illustrates that new applications may present opportunities for entirely novel covert channels.

A key advantage of this channel for users is that it cannot be eliminated. In Chapter 7 we showed that while detection is non-trivial, the channel can be detected with accuracies of over 95% if the warden knows its modulation technique. However, even if the channel is detected receivers may still be partly protected, since it is a broadcast channel.

In Chapter 6 we analysed an indirect noisy temperature-based timing channel. First we developed a novel technique to reduce a main component of the channel noise by up to two orders of magnitude. We then proposed a method to estimate the channel capacity. For two example intermediate hosts we found the capacity is only 10–20 bits per hour. Compared to other channels the capacity is very small and hence these channels are not well suited for general-purpose communication.

However, temperature-based channels are potentially hard to detect and eliminate, and can be used in situations where other channels are not available. Furthermore, they are very effective for revealing hidden services in anonymisation networks. We proposed several measures to reduce the channel's capacity.

We demonstrated that machine-learning techniques are effective in detecting different types of covert channels with accuracies of over 95% based on different characteristics, some of which we developed specifically for particular channels.

Based on our analysis of different channels we draw the following conclusions. Many existing modulation schemes were developed ad-hoc, without good knowledge of the characteristics of normal traffic. For noise-free channels this may not have consequences, but for noisy channels this can result in significantly reduced capacity and stealth.

Noisy covert channels provide sufficient throughput for communication purposes. Ideally reliable transport techniques should exploit information of overt protocols to minimise packet loss and reordering related channel errors. However, we demonstrated that even more general reliable transport techniques provide adequate performance.

Our work characterises the trade-offs between channel complexity, capacity, robustness and stealth. Simpler channels are easier to implement and have higher capacities but on the other hand are also easier to eliminate and detect. More complex channels are more difficult to implement and have lower capacities, but are harder to eliminate and detect. Indirect covert channels provide additional security, but are harder to construct and have lower capacity than direct channels.

8.1 Future research directions

This thesis raises a number of new avenues for future research. Besides extending our study of the different channels towards a wider range of operating conditions, for each of the topics explored in this thesis there remain a number of items for further research.

The TTL channel's stealth can be improved through modulation schemes using more sophisticated models of regular TTL variation. Our proposed technique for reliable data transport, which ought to be applicable to other network-layer storage and timing channels, can be improved to increase throughput with higher noise levels.

Our improved inter-packet timing encoding schemes can be used for both active and passive channels, but we have not evaluated the performance for active channels. Also, we have not investigated encoding more than one bit per inter-packet gap. The buffering delay management algorithm should be improved to minimise the added delay for passive channels. How to automatically select the optimal model for sub-band encoding is another open question. Our implementation could be further improved to reduce noise introduced at the sender and receiver. Another interesting area for research is the study of other timing channels, such as the timing of message sequences.

There is a lot of room to further explore the novel game-traffic channel. The channel could be studied in more detail in the context of the particular game we used (more players, different maps and game settings). The encoding scheme could be refined to improve the stealth and to increase the throughput, especially for large round trip times. Furthermore, the channel could be applied to other games or immersive virtual worlds, where player character movements are regularly propagated to other participants. However, the important point of this research is to show that new applications may offer completely novel channels.

A number of aspects of the temperature-based covert channel remain unexplored. First and foremost we have not analysed the channel's capacity for remote covert senders modulating CPU load through a varying rate of requests. Furthermore, we did not study how information can be transmitted reliably across the channel. Besides covert communications, the channel could be used for approximate geo-location of hosts based on remotely measured daily temperature patterns. The development and analysis of such techniques remains future work. Apart from our work, indirect channels have been barely analysed and should receive more attention.

We proposed elimination techniques for the different channels, but we have not implemented and analysed all of them. Most notable, the design and analysis of techniques for eliminating inter-packet gap timing channels needs further work. Other noise distributions or noise-introducing techniques that limit the channel capacity with minimum side-effects for different types of traffic should be evaluated. Adaptive techniques are needed that introduce noise at different levels based on the characteristics of overt traffic.

We showed that most channels are detected with relatively high accuracy, but there is still room for improvement. New research could focus on improving the detection accuracy by developing new features or using different machine-learning algorithms. Methods for detecting temperature-based covert channels have not been studied yet.

The software framework for creating covert channels could be extended with new covert channel techniques and more efficient schemes for reliable transport. Currently there is no support for link-layer channels and there are no techniques for reliable transmission based on retransmissions. In the future the software could be modified so that it also becomes a framework for detecting and eliminating covert channels, making it possible to analyse and compare the efficiency of different countermeasures.

Very little work exists on formal methods for the identification of covert channels in network protocols. Instead of having to eliminate or detect existing covert channels at least some of them should be prevented during protocol design. However, with a lack of formal identification methods, covert channel identification remains ad-hoc at best or is not attempted at all in the worst case.

The construction of high-capacity covert channels that are hard to detect and eliminate is challenging. This is made even more difficult since error detection and correction mechanisms need to be used over noisy channels. Unless well hidden such mechanisms can reveal the channel.

We think that new applications and higher protocol layers still offer many unexplored opportunities for covert channels. Furthermore, different types of channels could be combined to increase the performance. On the other hand the development of improved channels necessitates further work to develop improved countermeasures. It seems likely that the arms race of developing new improved covert channels and developing more effective countermeasures will continue.

Bibliography

[1] S. Zander, G. Armitage, P. Branch. Reliable Transmission Over Covert Channels in First Person Shooter Multiplayer Games. In *Proceedings of 34th Annual IEEE Conference on Local Computer Networks (LCN)*, October 2009.

[2] S. Zander, G. Armitage, P. Branch. Covert Channels in Multiplayer First Person Shooter Online Games. In *Proceedings of 33rd Annual IEEE Conference on Local Computer Networks (LCN)*, October 2008.

[3] S. Zander, S. J. Murdoch. An Improved Clock-skew Measurement Technique for Revealing Hidden Services. In *Proceedings of Usenix Security*, July/August 2008.

[4] S. Zander, G. Armitage, P. Branch. Covert Channels and Countermeasures in Computer Network Protocols. *IEEE Communications Magazine*, 45(12):136–142, December 2007.

[5] S. Zander, G. Armitage, P. Branch. An Empirical Evaluation of IP Time To Live Covert Channels. In *Proceedings of IEEE International Conference on Networks (ICON)*, November 2007.

[6] S. Zander, G. Armitage, P. Branch. A Survey of Covert Channels and Countermeasures in Computer Network Protocols. *IEEE Communications Surveys and Tutorials*, 9(3):44–57, October 2007.

[7] S. Zander, P. Branch, G. Armitage. Error Probability Analysis of IP Time To Live Covert Channels. In *Proceedings of IEEE International Symposium on Communications and Information Technologies (ISCIT)*, October 2007.

[8] S. Zander, G. Armitage, P. Branch. Covert Channels in the IP Time To Live Field. In *Proceedings of Australian Telecommunication Networks and Applications Conference (ATNAC)*, December 2006.

[9] S. Zander, G. Armitage, P. Branch. Dynamics of the IP Time To Live Field in Internet Traffic Flows. Technical Report 070529A, CAIA Technical Report, May 2007. `http://caia.swin.edu.au/reports/070529A/CAIA-TR-070529A.pdf`.

[10] S. Zander, G. Armitage. CCHEF – Covert Channels Evaluation Framework Design and Implementation. Technical Report 080530A, CAIA Technical Report, May 2008. `http://caia.swin.edu.au/reports/080530A/CAIA-TR-080530A.pdf`.

[11] P. Branch, S. Zander. Bit-Stuffing Rate in the High-Level Data Link Control (HDLC) Protocol. Technical Report 081121A, CAIA Technical Report, November 2008. `http://caia.swin.edu.au/reports/081121A/CAIA-TR-081121A.pdf`.

[12] S. Zander, P. Branch, G. Armitage. Estimating the Capacity of Temperature-based Covert Channels. Technical Report 091218A, CAIA Technical Report, December 2009. `http://caia.swin.edu.au/reports/091218A/CAIA-TR-091218A.pdf`.

[13] B. Lampson. A Note on the Confinement Problem. *Communication of the ACM*, 16(10):613–615, October 1973.

[14] V. Gligor. A Guide to Understanding Covert Channel Analysis of Trusted Systems. Technical Report NCSC-TG-030, National Computer Security Center, November 1993. `http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-030.html`.

[15] C. G. Girling. Covert Channels in LAN's. *IEEE Transactions on Software Engineering*, SE-13(2):292–296, February 1987.

[16] T. Handel, M. Sandford. Hiding Data in the OSI Network Model. In *Proceedings of the First International Workshop on Information Hiding*, pages 23–38, 1996.

[17] F. A. P. Petitcolas, R. J. Anderson, M. G. Kuhn. Information Hiding – A Survey. *Proceedings of the IEEE, special issue on protection of multimedia content*, 87(7):1062–1078, July 1999.

[18] J. Millen. 20 Years of Covert Channel Modeling and Analysis. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 113–114, May 1999.

[19] US Department of Defense Standard. Trusted Computer System Evaluation Criteria. Technical Report DOD 5200.28-STD, US Department of Defense, December 1985. `http://csrc.nist.gov/publications/history/dod85.pdf`.

[20] M. Mitzenmacher. A Survey of Results for Deletion Channels and Related Synchronization Channels. *Probability Surveys*, 6:1–33, 2009.

[21] C. E. Shannon. A Mathematical Theory of Communications. *Bell Systems Technical Journal*, 27(3), July 1948.

[22] T. M. Cover, J. A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley & Sons, 1991.

[23] Office for Official Publications of the European Communities. Information Technology Security Evaluation Criteria (ITSEC): Preliminary Harmonised Criteria - Version 1.2, June 1991. `http://www.ssi.gouv.fr/site_documents/ITSEC/ITSEC-uk.pdf`.

[24] The Common Criteria Project. Common Criteria for Information Technology Security Evaluation Part 3. Technical Report CCIMB-2004-01-003, January 2004. `http://www.commoncriteriaportal.org/public/files/ccpart3v2.2.pdf`.

[25] Australian Government, Department of Defence. About the Evaluated Product List (EPL), July 2007. `http://www.dsd.gov.au/infosec/evaluation_services/epl/AboutEPL.html`.

[26] K. Maney. Bin Laden's Messages Could Be Hiding In Plain Sight. *USA Today*, December 2001. `http://www.usatoday.com/life/cyber/ccarch/2001/12/19/maney.htm`.

[27] N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan. Infranet: Circumventing Web Censorship and Surveillance. In *Proceedings of 11th USENIX Security Symposium*, August 2002.

[28] Linden Research, Inc. Second Life – How Corporations Use Virtual Worlds. `http://secondlifegrid.net/slfe/corporations-use-virtual-world`.

[29] C. Tsai, V. D. Gligor, C. S. Chandersekaran. A Formal Method for the Identification of Covert Storage Channels in Source Code. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.

[30] R. A. Kemmerer. A Practical Approach to Identifying Storage and Timing Channels. In *Proceedings of IEEE Symposium on Security and Privacy*, April 1982.

[31] G. J. Simmons. The Prisoners' Problem and the Subliminal Channel. In *Proceedings of Advances in Cryptology (CRYPTO)*, pages 51–67, 1983.

[32] S. Craver. On Public-Key Steganography in the Presence of an Active Warden. In *Proceedings of the Second International Workshop on Information Hiding*, pages 355–368, April 1998.

[33] N. Lucena, J. Pease, P. Yadollahpour, S. J. Chapin. Syntax and Semantics-Preserving Application-Layer Protocol Steganography. In *Proceedings of 6th Information Hiding Workshop*, May 2004.

[34] BBC News. Airlines bomb plot: The e-mails, September 2009. `http://news.bbc.co.uk/2/hi/uk_news/8193501.stm`.

[35] G. Shah, A. Molina, M. Blaze. Keyboards and Covert Channels. In *Proceedings of USENIX Security Symposium*, August 2006.

[36] C. H. Rowland. Covert Channels in the TCP/IP Protocol Suite. *First Monday, Peer Reviewed Journal on the Internet*, July 1997.

[37] D. V. Forte, C. Maruti, M. R. Vetturi, M. Zambelli. SecSyslog: An Approach to Secure Logging Based on Covert Channels. In *Proceedings of First International Workshop on Systematic Approaches to Digital Forensic Engineering*, pages 248–263, November 2005.

[38] The Honeynet Project. Know Your Enemy: Sebek - A Kernel Based Data Capture Tool. Technical report, 2003. `http://www.honeynet.org/papers/sebek.pdf`.

[39] S. R. White. Covert Distributed Processing with Computer Viruses. In *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 616–619, 1989.

[40] I. Moskowitz, R. Newman, P. Syverson. Quasi-Anonymous Channels. In *Proceedings of Communication, Network, and Information Security (CNIS)*, December 2003.

[41] J. Xu, J. Fan, M. Ammar, S. Moon. Prefixpreserving IP Address Anonymization: Measurement-based Security Evaluation and a New Cryptography-based Scheme. In *Proceedings of the 10th IEEE International Conference on Network Protocols*, November 2002.

[42] J. Bethencourt, J. Franklin, M. Vernon. Mapping Internet Sensors with Probe Response Attacks. In *Proceedings of USENIX Security Symposium*, August 2005.

[43] S. J. Murdoch, G. Danezis. Low-cost traffic analysis of Tor. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 183–195, May 2005.

[44] S. J. Murdoch. Hot or Not: Revealing Hidden Services by Their Clock Skew. In *Proceedings of 13th ACM Conference on Computer and Communications Security (CCS)*, pages 27–36, November 2006.

[45] R. deGraaf, J. Aycock, M. Jacobson Jr. Improved Port Knocking with Strong Authentication. In *Proceedings of 21st Annual Computer Security Applications Conference*, December 2005.

[46] W. Mazurczyk, Z. Kotulski. New Security and Control Protocol for VoIP Based on Steganography and Digital Watermarking. Technical report, Institute of Fundamental Technological Research, Polish Academy of Sciences, June 2005. `http://arxiv.org/ftp/cs/papers/0602/0602042.pdf`.

[47] W. Mazurczyk, Z. Kotulski. New VoIP Traffic Security Scheme with Digital Watermarking. In *Proceedings of International Conference on Computer Safety, Reliability, and Security (SafeComp)*, pages 170–181, September 2006.

[48] E. Jones, O. Le Moigne, J.-M. Robert. IP Traceback Solutions Based on Time to Live Covert Channel. In *Proceedings of 12th IEEE International Conference on Networks (ICON)*, pages 451–457, November 2004.

[49] H. Qu, Q. Cheng, E. Yaprak. Using Covert Channel to Resist DoS Attacks in WLAN. In *Proceedings of International Conference on Wireless Networks*, pages 38–44, June 2005.

[50] Z. Wang, J. Deng, R. B. Lee. Mutual Anonymous Communications: A New Covert Channel Based on Splitting Tree MAC. In *Proceedings of IEEE INFOCOM*, May 2007.

[51] A. Houmansadr, N. Kiyavash, N. Borisov. RAINBOW: A Robust And Invisible Non-Blind Watermark for Network Flows. In *Proceedings of 16th Annual Network & Distributed System Security Symposium (NDSS)*, February 2009.

[52] N. E. Proctor, P. G. Neumann. Architectural Implications of Covert Channels. In *Proceedings of the 15th National Computer Security Conference*, pages 28–43, October 1992.

[53] A. B. Jeng, M. D. Abrams. On Network Covert Channel Analysis. In *Proceedings of Third Aerospace Computer Security Conference*, December 1987.

[54] I. S. Moskowitz, M. H. Kang. Covert Channels – Here to Stay? In *Proceedings of 9th Annual Conference on Computer Assurance*, pages 235–244, 1994.

[55] D. Kundur, K. Ahsan. Practical Internet Steganography: Data Hiding in IP. In *Proceedings of Texas Workshop on Security of Information Systems*, April 2003.

[56] A. Hintz. Covert Channels in TCP and IP Headers, 2003. `http://www.defcon.org/images/defcon-10/dc-10-presentations/dc10-hintz-covert.ppt`.

[57] J. Postel. User Datagram Protocol. RFC 0768, IETF, August 1980. `http://www.ietf.org/rfc/rfc0768.txt`.

[58] G. Fisk, M. Fisk, C. Papadopoulos, J. Neil. Eliminating Steganography in Internet Traffic with Active Wardens. In *Proceedings of 5th International Workshop on Information Hiding*, October 2002.

[59] Audio-Video Transport Working Group, H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, IETF, January 1996. `http://www.ietf.org/rfc/rfc1889.txt`.

[60] T. Graf. Messaging over IPv6 Destination Options. Technical report, Swiss Unix User Group, 2003. `http://gray-world.net/papers/messip6.txt`.

[61] N. B. Lucena, G. Lewandowski, S. J. Chapin. Covert Channels in IPv6. In *Proceedings of Privacy Enhancing Technologies (PET)*, pages 147–166, May 2005.

[62] Z. Trabelsi, H. El-Sayed, L. Frikha, T. F. Rabie. Traceroute Based IP Channel for Sending Hidden Short Messages. In *Proceedings of Advances in Information and Computer Security (IWSEC)*, pages 421–436, October 2006.

[63] M. Wolf. Covert Channels in LAN Protocols. In *Proceedings of the Workshop on Local Area Network Security (LANSEC)*, pages 91–101, 1989.

[64] J. Giffin, R. Greenstadt, P. Litwack, R. Tibbetts. Covert Messaging Through TCP Timestamps. In *Proceedings of the Privacy Enhancing Technologies Workshop (PET)*, pages 194–208, April 2002.

[65] M. A. Padlipsky, D. W. Snow, P. A. Karger. Limitations of End-to-End Encryption in Secure Computer Networks. Technical Report ESD-TR-78-158, Mitre Corporation, August 1978. `http://stinet.dtic.mil/cgi-bin/GetTRDoc?AD=A059221&Location=U2&doc=GetTRDoc.pdf`.

[66] A. Galatenko, A. Grusho, A. Kniazev, E. Timonina. Statistical Covert Channels Through PROXY Server. In *Proceedings of Third International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 424–429, September 2005.

[67] L. Bowyer. Firewall Bypass via Protocol Steganography, September 2002. `http://www.networkpenetration.com/protocol_steg.html`.

[68] B. Yuan, P. Lutz. A Covert Channel in Packet Switching Data Networks. In *Proceedings of The Second Upstate New York Workshop on Communications and Networking*, November 2005.

[69] L. Ji, W. Jiang, B. Dai, X. Niu. A Novel Covert Channel Based on Length of Messages. In *Proceedings of International Symposium on Information Engineering and Electronic Commerce*, pages 551–554, 2009.

[70] M. C. Perkins. Hiding out in Plaintext: Covert Messaging with Bitwise Summations. Master's thesis, Iowa State University, 2005.

[71] M. Marone. Adaptation and Performance of Covert Channels in Dynamic Source Routing. Technical report, Computer Science Department, Yale University, December 2003. `http://zoo.cs.yale.edu/classes/cs490/03-04a/michael.marone/paper.pdf`.

[72] S. Li, A. Ephremides. A Network Layer Covert Channel in Ad-hoc Wireless Networks. In *Proceedings of First IEEE Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, pages 88–96, October 2004.

[73] C. Krätzer, J. Dittmann, A. Lang, T. Kühne. WLAN Steganography: A First Practical Review. In *Proceedings of 8th ACM Multimedia and Security Workshop*, September 2006.

[74] A. Dyatlov, S. Castro. Exploitation of Data Streams Authorized by a Network Access Control System for Arbitrary Data Transfers: Tunneling and Covert Channels over the HTTP Protocol. Technical report, Gray-World, June 2003. `http://gray-world.net/projects/papers/covert_paper.txt`.

[75] Z. Kwecka. Application Layer Covert Channel Analysis and Detection. Technical report, Napier University Edinburgh, 2006. `http://www.buchananweb.co.uk/zk.pdf`.

[76] M. Van Horenbeeck. Deception on the Network: Thinking Differently About Covert Channels. In *Proceedings of 7th Australian Information Warfare and Security Conference*, December 2006.

[77] S. Castro and Gray World Team. Cooking Channels. *hakin9 Magazine (www.hakin9.org)*, pages 50–57, May 2006.

[78] L. Y. Bai, Y. Huang, G. Hou, B. Xiao. Covert Channels Based on Jitter Field of the RTCP Header. In *Proceedings of International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 1388–1391, 2008.

[79] K. Szczypiorski. HICCUPS: Hidden Communication System for Corrupted Networks. In *Proceedings of 10th International Multi-Conference on Advanced Computer Systems*, pages 31–40, October 2003.

[80] L. Butti, F. Veysset. Wi-Fi Advanced Stealth. In *Proceedings of Black Hat US*, August 2006. `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Veyssett.pdf`.

[81] M. Smeets, M. Koot. Research Report: Covert Channels. Master's thesis, University of Amsterdam, February 2006.

[82] daemon9. LOKI2: The Implementation. *Phrack Magazine*, 7(51), September 1997.

[83] D. Stødle. ptunnel – Ping Tunnel, 2005. `http://www.cs.uit.no/daniels/PingTunnel`.

[84] B. Ray, S. Mishra. Secure and Reliable Covert Channel. In *Proceedings of the 4th Annual Workshop on Cyber Security and Information Intelligence Research*, pages 1–3, 2008.

[85] P. Padgett. Corkscrew, 2001. `http://www.agroman.net/corkscrew/`.

[86] A. Dyatlov. Firepass – Is a Tunneling Tool, 2003. `http://gray-world.net/pr_firepass.shtml`.

[87] P. LeBoutillier. HTTunnel, 2005. `http://sourceforge.net/projects/httunnel/`.

[88] L. Nussbaum, P. Neyron, O. Richard. On Robust Covert Channels Inside DNS. In *Proceedings of IFIP Advances in Information and Communication Technology*, pages 51–62, June 2009.

[89] J. Postel. Internet Protocol. RFC 0791, IETF, September 1981. `http://www.ietf.org/rfc/rfc0791.txt`.

[90] K. Ahsan, D. Kundur. Practical Data Hiding in TCP/IP. In *Proceedings of ACM Workshop on Multimedia Security*, December 2002.

[91] E. Cauich, R. Gómez Cárdenas, R. Watanabe. Data Hiding in Identification and Offset IP Fields. In *Proceedings of 5th International School and Symposium of Advanced Distributed Systems (ISSADS)*, pages 118–125, January 2005.

[92] J. Postel. Transmission Control Protocol. RFC 0793, IETF, September 1981. `http://www.ietf.org/rfc/rfc0793.txt`.

[93] J. Rutkowska. The Implementation of Passive Covert Channels in the Linux Kernel. In *Proceedings of Chaos Communication Congress*, December 2004.

[94] S. J. Murdoch, S. Lewis. Embedding Covert Channels into TCP/IP. In *Proceedings of 7th Information Hiding Workshop*, June 2005.

[95] H. Qu, P. Su, D. Feng. A Typical Noisy Covert Channel in the IP Protocol. In *Proceedings of 38th Annual International Carnahan Conference on Security Technology*, pages 189–192, October 2004.

[96] I. Zelenchuk. Skeeve – ICMP bounce tunnel, 2004. `http://www.gray-world.net/poc_skeeve.shtml`.

[97] G. Danezis. Covert Communications Despite Traffic Data Retention. Technical report, ESAT, University of Leuven, January 2005. `http://homes.esat.kuleuven.be/~gdanezis/cover.pdf`.

[98] Anonymous. DNS Covert Channels and Bouncing Techniques, 2005. `http://archives.neohapsis.com/archives/fulldisclosure/2005-07/att-0472/p63_dns_worm_covert_channel.txt`.

[99] M. Bauer. New Covert Channels in HTTP: Adding Unwitting Web Browsers to Anonymity Sets. In *Proceedings of Workshop On Privacy In The Electronic Society*, pages 72–78, October 2003.

[100] S. Cabuk, C. E. Brodley, C. Shields. IP Covert Timing Channels: Design and Detection. In *Proceedings of 11th ACM Conference on Computer and Communications Security (CCS)*, pages 178–187, October 2004.

[101] L. Yao, X. Zi, L. Pan, J. Li. A Study of On/off Timing Channel Based on Packet Delay Distribution. *Computers & Security*, In Press, Corrected Proof, 2009.

[102] X. Luo, E. W. W. Chan, R. K. C. Chang. TCP Covert Timing Channels: Design and Detection. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2008.

[103] V. Berk, A. Giani, G. Cybenko. Detection of Covert Channel Encoding in Network Packet Delays. Technical Report TR2005-536, Department of Computer Science, Dartmouth College, November 2005. `http://www.ists.dartmouth.edu/library/149.pdf`.

[104] S. Gianvecchio, H. Wang, D. Wijesekera, S. Jajodia. Model-Based Covert Timing Channels: Automated Modeling and Evasion. In *Proceedings of Recent Advances in Intrusion Detection (RAID) Symposium*, September 2008.

[105] S. H. Sellke, C.-C. Wang, S. Bagchi, N. B. Shroff. Covert TCP/IP Timing Channels: Theory to Implementation. In *Proceedings of the 28th Conference on Computer Communications (INFOCOM)*, April 2009.

[106] Y. Liu, D. Ghosal, F. Armknecht, A.-R. Sadeghi, S. Schulz, S. Katzenbeisser. Hide and Seek in Time – Robust Covert Timing Channels. In *Proceedings of 14th European Symposium on Research in Computer Security*, September 2009.

[107] S. Gianvecchio, H. Wang. Detecting Covert Timing Channels: An Entropy-Based Approach. In *Proceedings of 14th ACM Conference on Computer and Communication Security (CCS)*, November 2007.

[108] X. Y. Wang, S. Chen, S. Jajodia. Tracking Anonymous Peer-to-Peer VoIP Calls on the Internet. In *Proceedings of the 12th ACM Conference on Computer Communications Security (CCS)*, November 2005.

[109] X. Wang, S. Chen, S. Jajodia. Network Flow Watermarking Attack on Low-latency Anonymous Communication Systems. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 116–130, 2007.

[110] H.-G. Eßer, F. C. Freiling. Kapazitätsmessung eines verdeckten Zeitkanals über HTTP. Technical Report TR-2005-10, Universität Mannheim, 2005. `http://bibserv7.bib.uni-mannheim.de/madoc/volltexte/2005/1136/pdf/tr_2005_10.pdf` (in german).

[111] X. Zou, Q. Li, S. Sun, X. Niu. The Research on Information Hiding Based on Command Sequence of FTP Protocol. In *Proceedings of 9th International Conference on Knowledge-Based Intelligent Information and Engineering Systems*, pages 1079–1085, September 2005.

[112] J. Postel and J.K. Reynolds. File Transfer Protocol. RFC 0959, IETF, October 1985. `http://www.ietf.org/rfc/rfc0959.txt`.

[113] S. D. Servetto, M. Vetterli. Communication Using Phantoms: Covert Channels in the Internet. In *Proceedings of IEEE International Symposium on Information Theory (ISIT)*, June 2001.

[114] W. Mazurczyk, M. Smolarczyk, K. Szczypiorski. Hiding Information in Retransmissions. *Cryptography and Security*, abs/0905.0363, 2009.

[115] R. C. Chakinala, A. Kumarasubramanian, R. Manokaran, G. Noubir, C. Pandu Rangan, R. Sundaram. Steganographic Communication in Ordered Channels. In *Proceedings of 8th International Workshop on Information Hiding*, pages 42–57, July 2006.

[116] X. Luo, E. W. W. Chan, R. K. C. Chang. Cloak: A Ten-fold Way for Reliable Covert Communications. In *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, September 2007.

[117] H. Khan, Y. Javed, S. A. Khayam, F. Mirza. Embedding a Covert Channel in Active Network Connections. In *Proceedings of IEEE Global Communications Conference (GlobeCom)*, December 2009.

[118] A. El-Atawy, E. Al-Shaer. Building Covert Channels over the Packet Reordering Phenomenon. In *Proceedings of 28th Annual IEEE Conference on Computer Communications (INFOCOM)*, 2009.

[119] S. Bhadra, S. Shakkottai, S. Vishwanath. Covert Communication over Slotted ALOHA Systems. In *Proceedings of the 42nd Allerton Conference on Communication, Control, and Computing*, October 2004.

[120] T. M. Dogu, A. Ephremides. Covert Information Transmission through the Use of Standard Collision Resolution Algorithms. In *Proceedings of the Third International Workshop on Information Hiding (IH)*, pages 419–433, September 1999.

[121] S. Li, A. Ephremides. A Covert Channel in MAC Protocols based on Splitting Algorithms. In *Proceedings of Wireless Communications and Networking Conference (WCNC)*, pages 1168–1173, March 2005.

[122] S. J. Murdoch, P. Zielinski. Covert Channels for Collusion in Online Computer Games. In *Proceedings of 6th Information Hiding Workshop*. Springer, May 2004.

[123] J. C. Hernandez-Castro, I. Blasco-Lopez, J. M. Estevez-Tapiador, A. Ribagorda-Garnacho. Steganography in Games: A General Methodology and its Application to the Game of Go. *Computers & Security*, 25(1):64–71, February 2006.

[124] M. Diehl. Secure Covert Channels in Multiplayer Games. In *Proceedings of the 10th ACM Workshop on Multimedia and Security*, pages 117–122, 2008.

[125] A. Desoky, M. Younis. Chestega: Chess Steganography Methodology. *Security and Communication Networks*, 2(6):555–566, 2009.

[126] D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[127] J. Millen. Information Flow Analysis of Formal Specifications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 3–8, April 1981.

[128] J. A. Goguen, J. Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[129] R. A. Kemmerer. Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels. *ACM Transactions on Computer Systems (TOCS)*, 1(3):256–277, August 1983.

[130] R. A. Kemmerer. A Practical Approach to Identifying Storage and Timing Channels: Twenty Years Later. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, pages 109–118, December 2002.

[131] R. Kemmerer, P. Porras. Covert Flow Trees: A Visual Approach to Analyzing Covert Storage Channels. *IEEE Transactions on Software Engineering*, SE-17(11):1166–1185, November 1991.

[132] A. L. Donaldson, J. McHugh, K. A. Nyberg. Covert Channels in Trusted LANs. In *Proceedings of the 11th NBS/NCSC National Computer Security Conference*, pages 226–232, October 1988.

[133] L. Hélouët, C. Jard, M. Zeitoun. Covert Channels Detection in Protocols Using Scenarios. In *Proceedings of Workshop on Security Protocols Verification (SPV)*, April 2003.

[134] A. Aldini, M. Bernardo. An Integrated View of Security Analysis and Performance Evaluation: Trading QoS with Covert Channel Bandwidth. In *Proceedings of 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 283–296, September 2004.

[135] G. R. Malan, D. Watson, F. Jahanian, P. Howell. Transport and Application Protocol Scrubbing. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, pages 1381–1390, March 2000.

[136] M. Handley, C. Kreibich, V. Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of 10th USENIX Security Symposium*, August 2001.

[137] A. Singh, O. Nordström, C. Lu, A. L. M. dos Santos. Malicious ICMP Tunneling: Defense against the Vulnerability. In *Proceedings of 8th Australasian Conference on Information Security and Privacy (ACISP)*, pages 226–235, July 2003.

[138] N. Schear, C. Kintana, Q. Zhang, A. Vahdat. Glavlit: Preventing Exfiltration at Wire Speed. In *Proceedings of Fifth Workshop on Hot Topics in Networks (HotNets)*, November 2006.

[139] J. K. Millen. Covert Channel Capacity. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 60–66, May 1987.

[140] I. S. Moskowitz, A. R. Miller. Simple Timing Channels. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 56–64, 1994.

[141] H. Wei-Ming. Reducing Timing Channels with Fuzzy Time. In *Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy*, pages 8–20, May 1991.

[142] J. W. Gray III. Countermeasures and Tradeoffs for a Class of Covert Timing Channels. Technical Report HKUST-CS94-18, Hong Kong University of Science and Technology, 2000. `http://repository.ust.hk/dspace/bitstream/1783.1/25/1/tr94-18.pdf`.

[143] R. E. Newman-Wolfe, B. R. Venkatraman. High Level Prevention of Traffic Analysis. In *Proceedings of Seventh Annual Computer Security Applications Conference*, pages 102–109, December 1991.

[144] B. R. Venkatraman, R. E. Newman-Wolfe. Transmission Schedules To Prevent Traffic Analysis. In *Proceedings of 9th Annual Computer Security and Applications Conference*, pages 108–115, December 1993.

[145] B. Graham, Y. Zhu, X. Fu, R. Bettati. Using Covert Channels to Evaluate the Effectiveness of Flow Confidentiality Measures. In *Proceedings of 11th International Conference on Parallel and Distributed Systems*, pages 57–63, July 2005.

[146] J. Giles, B. Hajek. The Jamming Game for Packet Timing Channels. In *Proceedings of IEEE International Symposium on Information Theory (ISIT)*, June 2000.

[147] Y. Wang, P. Chen, Y. Ge, B. Mao, L. Xie. Traffic Controller: A Practical Approach to Block Network Covert Timing Channel. In *Proceedings of International Conference on Availability, Reliability and Security*, pages 349–354, 2009.

[148] D. Bell, L. LaPadula. Secure Computer Systems: Mathematical Foundation. Technical Report ESD-TR-73-278, Mitre Corp, 1973.

[149] N. Ogurtsov, H. Orman, R. Schroeppel, S. O'Malley, O. Spatscheck. Experimental Results of Covert Channel Limitation in One-Way Communication Systems. In *Proceedings of Symposium on Network and Distributed System Security (SNDSS)*, February 1997.

[150] M. H. Kang, I. S. Moskowitz. A Pump for Rapid, Reliable, Secure Communication. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 119–129, 1993.

[151] M. H. Kang, I. S. Moskowitz, D. C. Lee. A Network Version of the Pump. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 144–154, May 1995.

[152] R. W. Smith, G. S. Knight. Predictable Design of Network-Based Covert Communication Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 311–321, 2008.

[153] T. Sohn, J. Seo, J. Moon. A Study on the Covert Channel Detection of TCP/IP Header Using Support Vector Machine. In *Proceedings of 5th International Conference on Information and Communications Security*, pages 313–324, October 2003.

[154] E. Tumoian, M. Anikeev. Network Based Detection of Passive Covert Channels in TCP/IP. In *Proceedings of 1st IEEE LCN Workshop on Network Security*, pages 802–809, November 2005.

[155] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, IETF, May 1992. `http://www.ietf.org/rfc/rfc1323.txt`.

[156] B. R. Venkatraman, R. E. Newman-Wolfe. Capacity Estimation and Auditability of Network Covert Channels. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 186–198, May 1995.

[157] R. M. Stillman. Detecting IP Covert Timing Channels by Correlating Packet Timing With Memory Content. In *Proceedings of IEEE SoutheastCon*, pages 204–209, 2008.

[158] T. Sohn, J. Moon, S. Lee, D. H. Lee, J. Lim. Covert Channel Detection in the ICMP Payload Using Support Vector Machine. In *Proceedings of 18th International Symposium on Computer and Information Sciences (ISCIS)*, pages 828–835, November 2003.

[159] D. Pack, W. Streilein, S. E. Webster, R. K. Cunningham. Detecting HTTP Tunneling Activities. In *Proceedings of Third Annual Information Assurance Workshop*, June 2002.

[160] K. Borders, A. Prakash. Web Tap: Detecting Covert Web Traffic. In *Proceedings of 11th ACM Conference on Computer and Communications Security (CCS)*, pages 110–120, October 2004.

[161] A. Fei, G. Pei, R. Liu, L. Zhang. Measurements on Delay and Hop-Count of the Internet. In *Proceedings of IEEE GLOBECOM - Internet Mini-Conference*, 1998.

[162] F. Begtasevic, P. van Mieghen. Measurements of the Hop Count in the Internet. In *Proceedings of Workshop on Passive and Active Measurement (PAM)*, pages 183–190, April 2001.

[163] B. Huffaker, M. Fomenkov, D. Moore, K. Claffy. Macroscopic Analyses of the Infrastructure: Measurement and Visualization of Internet Connectivity and Performance. In *Proceedings of Passive and Active Measurement Workshop (PAM)*, April 2001.

[164] C. Shen, H. Schulzrinne, S. Lee and J. Bang. Routing Dynamics Measurement and Detection for Next Step Internet Signaling Protocol. In *Proceedings of IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services (E2EMON)*, May 2005.

[165] R. van de Meent. M2C Measurement Data Repository, December 2003. `http://m2c-a.cs.utwente.nl/repository/`.

[166] NLANR PMA: Special Traces Archive. `http://pma.nlanr.net/Special/`.

[167] C. Jin, H. Wang, K. Shin. Hop-Count Filtering: An Effective Defense Against Spoofed DoS Traffic. In *Proceedings of the 10th ACM International Conference on Computer and Communications Security (CCS)*, pages 30–41, October 2003.

[168] N. Davids. Initial TTL Values. `http://members.cox.net/~ndav1/self_published/TTL_values.html`.

[169] R. Silverman. On Binary Channels and Their Cascades. *IEEE Transactions on Information Theory*, 1(3), December 1955.

[170] J. Feng, Z. Ouyang, L. Xu, B. Ramamurthy. Packet Reordering in High-Speed Networks and Its Impact on High-Speed TCP Variants. In *Proceedings of the 5th International Workshop on Protocols for Fast Long-Distance Networks*, February 2007.

[171] R. G. Gallager. Sequential Decoding for Binary Channels With Noise and Synchronization Errors. Technical report, MIT Lincoln Lab, October 1961.

[172] K. S. Zigangirov. Sequential Decoding for a Binary Channel With Dropouts and Insertions. *Problems Information Transmission*, 5(2):17–22, 1969. Translated from Problemy Peredachi Informatsii, vol. 5, no. 2, pp. 23–30, 1969.

[173] D. Leigh. Capacity of Insertion and Deletion Channels. Technical report, July 2001. `http://www.inference.phy.cam.ac.uk/is/papers/leigh.ps`.

[174] J. Hu, T. M. Duman, M. F. Erden. On the Information Rates of Channels with Insertion/Deletion/Substitution Errors. In *Proceedings of International Conference on Communications (ICC)*, May 2008.

[175] S. Diggavi, M. Grossglauser. On Information Transmission over a Finite Buffer Channel. *IEEE Transactions on Information Theory*, 52(3):1226–1237, March 2006.

[176] D. Julian. Erasure networks. In *Proceedings of IEEE International Symposium on Information Theory*, page 138, June/July 2002.

[177] P. Karn. DSP and FEC Library, 2009. `http://www.ka9q.net/code/fec/`.

[178] T. C. Maxino, P. J. Koopman. The Effectiveness of Checksums for Embedded Control Networks. *IEEE Transactions on Dependable and Secure Computing*, 6(1), January/March 2009.

[179] E. A. Ratzer. Marker Codes for Channels With Insertions and Deletions. *Annals of Telecommunications*, 2005.

[180] E. A. Ratzer. Reliable Communication over Insertion-Deletion Channels. Technical report, Cambridge University Tripos Part III Project Report, 2000. `http://www.inference.phy.cam.ac.uk/ear23/papers/ptiii.pdf`.

[181] M. S. Borella, D. Swider, S. Uludag, G. B. Brewster. Internet Packet Loss: Measurement and Implications for End-to-End QoS. In *Proceedings of International Conference on Parallel Processing*, pages 3–12, 1998.

[182] V. Paxon. End-to-end Internet Packet Dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[183] Y. Zhang, N. Duffield, V. Paxson, S. Shenker. On the Constancy of Internet Path Properties. In *Proceedings of 1st ACM SIGCOMM Internet Measurement Workshop*, 2001.

[184] D. Loguinov, H. Radha. Measurement Study of Low-bitrate Internet Video Streaming. In *Proceedings of 1st ACM SIGCOMM Internet Measurement Workshop*, pages 281–293, 2001.

[185] J, Iannaccone, S. Jaiswal, C. Diot. Packet Reordering Inside the Sprint Backbone. Technical Report TR01-ATL-062917, Sprint ATL, 2001.

[186] Y. Wang, G. Lu, X. Li. A Study of Internet Packet Reordering. In *Proceedings of International Conference on Networking Technologies for Broadband and Mobile Networks (ICOIN)*, pages 350–359, February 2004.

[187] XMacro developers. XMacro. `http://xmacro.sourceforge.net`.

[188] Id Software. Quake 3. `http://www.idsoftware.com`.

[189] Linux Network Developers. Netem, 2008. `http://www.linuxfoundation.org/en/Net:Netem`.

[190] G. Armitage, M. Claypool, P. Branch. *Networking and Online Games - Understanding and Engineering Multiplayer Internet Games*. John Wiley & Sons, April 2006.

[191] Cisco Systems, Inc. Approaching the Zettabyte Era, 2008. `http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481374.pdf`.

[192] P. Branch, A. Heyde, G. Armitage. Rapid Identification of Skype Traffic. In *Proceedings of ACM Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, June 2009.

[193] T. Zseby, S. Zander, G. Carle. Evaluation of Building Blocks for Passive One-way Delay Measurements. In *Proceedings of Passive and Active Measurement (PAM) Workshop*, 2001.

[194] C. Henke, C. Schmoll, T. Zseby. Empirical Evaluation of Hash Functions for PacketID Generation in Sampled Multipoint Measurements. In *Proceedings of Passive and Active Measurement (PAM) Workshop*, pages 197–206, 2009.

[195] A. Cricenti, P. Branch. A Generalised Prediction Model of First Person Shooter Game Traffic. In *Proceedings of 34th Annual IEEE Conference on Local Computer Networks (LCN)*, October 2009.

[196] S. Gianvecchio. Timing Channels Proof-of-Concept Implementation. Available on request.

[197] Endace Limited. DAG Network Monitoring Cards. `http://www.endace.com/what-we-do/dag-network-monitoring-cards/`.

[198] T. Tso's, D. Hart (edts.). Real-Time Linux Wiki. `http://rt.wiki.kernel.org/index.php/Main_Page`.

[199] Y. Etsion, D. Tsafrir, Dror G. Feitelson. Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-time Processes. *SIGMETRICS Performance Evaluation Review*, 31(1):172–183, 2003.

[200] L. Rizo, D. Torres, J. Dehesa, D. Muñoz. Cauchy Distribution for Jitter in IP Networks. In *Proceedings of the 18th International Conference on Electronics, Communications and Computers*, pages 35–40, 2008.

[201] C. Demichelis and P. Chimento. IP Packet Delay Variation Metric for IP Performance Metrics (IPPM). RFC 3393, IETF, November 2002. `http://www.ietf.org/rfc/rfc3393.txt`.

[202] S. Zander, G. Armitage. Empirically Measuring the QoS Sensitivity of Interactive Online Game Players. In *Proceedings of Australian Telecommunications and Network Applications Conference (ATNAC)*, December 2004.

[203] Splash Damage. Enemy Territory: Return to Castle Wolfenstein. `http://www.enemy-territory.com`.

[204] G. Armitage. Optimising Online FPS Game Server Discovery through Clustering Servers by Origin Autonomous System. In *Proceedings of ACM Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, May 2008.

[205] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 2000.

[206] G. J. Armitage. An Experimental Estimation of Latency Sensitivity in Multiplayer Quake3. In *Proceedings of 11th IEEE International Conference on Networks (ICON)*, September 2003.

[207] M. Bond. Chewing on the 0xDEADBEEF, August 2007. `http://www.cl.cam.ac.uk/~mkb23/research/ChewingOnDeadBeef-Redact.pdf`.

[208] M. Abrash. *Graphics Programming Black Book, Chapter 64*. 2001. http://www.byte.com/abrash/chapters/gpbb64.pdf.

[209] A. Steed, C. Angus. Supporting Scalable Peer to Peer Virtual Environments Using Frontier Sets. In *Proceedings of IEEE Virtual Reality*, pages 27–34, March 2005.

[210] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, M. Claypool. The Effects of Loss and Latency on User Performance in Unreal Tournament 2003. In *Proceedings of ACM Network and System Support for Games (NetGames) Workshop*, August 2004.

[211]]]slug[[. SLUGBOT1.32 V19J – Quake 3 Client-side Bot. http://www.mpcdownloads.com/_mpc_d0wn_h4x_/Quake%201%202%203/q3unpure-slugbot13b.zip.

[212] C. Weidmann. Coding for the q-ary Symmetric Channel with Moderate q. In *Proceedings of IEEE International Symposium on Information Theory*, July 2008.

[213] S. Zander, D. Kennedy, G. Armitage. Dissecting Server-Discovery Traffic Patterns Generated By Multiplayer First Person Shooter Games. In *Proceedings of ACM Network and System Support for Games (NetGames) Workshop*, October 2005.

[214] R. Dingledine, N. Mathewson, P. F. Syverson. Tor: The Second-generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[215] D. M. Goldschlag, M. G. Reed, P. F. Syverson. Onion Routing. *Communications of the ACM*, 42(2):39–41, February 1999.

[216] BBC News. US blogger fired by her airline, November 2004. http://news.bbc.co.uk/1/technology/3974081.stm.

[217] Reporters Without Borders. Blogger and documentary filmmaker held for the past month, March 2006. http://www.rsf.org/article.php3?id_article=16810.

[218] L. Øverlier, P. F. Syverson. Locating Hidden Servers. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 100–114, May 2006.

[219] S. J. Murdoch, G. Danezis. Low-Cost Traffic Analysis of Tor. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 183–195, May 2005.

[220] S. B. Moon, P. Kelly, D. Towsley. Estimation and Removal of Clock Skew From Network Delay Measurements. Technical Report 98-43, Department of Computer Science, University of Massachusetts at Amherst, October 1998.

[221] T. Kohno, A. Broido, kc claffy. Remote Physical Device Fingerprinting. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 211–225, May 2005.

[222] D. Mills. Network Time Protocol (Version 3) Specification, Implementation. RFC 1305, IETF, March 1992. http://www.ietf.org/rfc/rfc1305.txt.

[223] C. Langton. Unlocking the Phase Lock Loop - Part 1, 2002. http://www.complextoreal.com/chapters/pll.pdf.

[224] Carnegie Mellon University. PID Tutorial. `http://www.engin.umich.edu/group/ctm/PID/PID.html`.

[225] MaxMind. GeoLite Country, 2008. `http://www.maxmind.com/app/geoip_country`.

[226] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[227] Apache Software Foundation. Apache Web Server. `http://www.apache.org/`.

[228] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3), July 2003.

[229] S. Clowes. TSOCKS - A Transparent SOCKS Proxying Library. `http://tsocks.sourceforge.net/`.

[230] The Mathworks. Simulink – Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink/`.

[231] R. Redelmeier. CPUBurn, June 2001. `http://pages.sbcglobal.net/redelm/`.

[232] W. S. Cleveland. LOWESS: A Program for Smoothing Scatterplots by Robust Locally Weighted Regression. *The American Statistician*, 35(1), 1981.

[233] S. Zander. Remote Synchronised Clock Skew Probing, 2008. `http://caia.swin.edu.au/cv/szander/cprobe/skew_probing.html`.

[234] T. Nguyen, G. Armitage. A Survey of Techniques for Internet Traffic Classification using Machine Learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008.

[235] N. Williams, S. Zander, G. Armitage. A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification. *SIGCOMM Computer Communication Review*, 36(5), October 2006.

[236] R. Kohavi, J. R. Quinlan. *Decision-tree Discovery*, chapter 16.1.3, pages 267–276. Oxford University Press, 2002.

[237] I. H. Witten, Eibe Frank. *"Data Mining: Practical Machine Learning Tools and Techniques – 2nd Edition*. Morgan Kaufmann, San Francisco, 2005.

[238] A. Porta, G. Baselli, D. Liberati, N. Montano, C. Cogliati, T. Gnecchi-Ruscone, A. Malliani, S. Cerutti. Measuring Regularity by Means of a Corrected Conditional Entropy in Sympathetic Outflow. *Biological Cybernetics*, 78(1), January 1998.

[239] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009.

[240] B. Veal, K. Li, D. Lowenthal. New Methods for Passive Estimation of TCP Round-Trip Times. In *Proceedings of Passive and Active Measurement (PAM) Workshop*, March/April 2005.

[241] S. Zander, G. Armitage, T. Nguyen, L. Mark, B. Tyo. Minimally Intrusive Round Trip Time Measurements Using Synthetic Packet-Pairs. Technical Report 060707A, CAIA Technical Report, July 2006. http://caia.swin.edu.au/reports/060707A/CAIA-TR-060707A.pdf.

[242] M. Dick, O. Wellnitz, L. Wolf. Analysis of Factors Affecting Players' Performance and Perception in Multiplayer Games. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames)*, pages 1–7, 2005.

[243] T. Lang, P. Branch, G. Armitage. A Synthetic Traffic Model for Quake 3. In *Proceedings of ACM SIGCHI International Conference on Advances in Computer Entertainment Technology (ACE)*, June 2004.

[244] S. J. Murdoch, S. Zander. Hot or Not: Fingerprinting Hosts Through Clock Skew. In *Presentation at EuroBSDCon*, September 2007. http://www.cl.cam.ac.uk/~sjm217/talks/eurobsdcon07hotornot.pdf.

[245] S. Zander. CCHEF – Covert Channels Evaluation Framework User Manual, May 2008. http://caia.swin.edu.au/cv/szander/cc/cchef/cchef-user-manual.pdf.

[246] D. A. Wheeler. SLOCCount – A Set of Tools for Counting Physical Source Lines of Code. http://www.dwheeler.com/sloccount/.

[247] S. Zander. CCHEF - Covert Channels Evaluation Framework, 2007. http://caia.swin.edu.au/cv/szander/cc/cchef/.

[248] H. Welte. Netfilter Queue Library. http://www.netfilter.org/projects/libnetfilter_queue/.

[249] WAND Network Research Group. libtrace – A Library for Trace Processing. http://research.wand.net.nz/software/libtrace.php.

[250] C. M. Grinstead, J. L. Snell. *Introduction to Probability: Second Revised Edition*. American Mathematical Society, 1997.

[251] R. Russell, H. Welte. Netfilter Hacking HOWTO. http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html.

[252] S. Zander, D. Kennedy, G. Armitage. KUTE – A High Performance Kernel-based UDP Traffic Engine. Technical Report 050118A, CAIA Technical Report, January 2005. http://caia.swin.edu.au/reports/050118A/CAIA-TR-050118A.pdf.

[253] mjrpes@yahoo.com. Quake 3 Map Compile Benchmark Page, 2008. http://ciole.net/quake_bench/.

[254] C. Croakin, P. Tobias (edts.). Engineering Statistics Handbook, Section 7.2.1.3. `http://www.itl.nist.gov/div898/handbook/prc/section2/prc213.htm`.

Appendices

Appendix A

Covert Channels Software Framework

To empirically evaluate covert channels according to the criteria outlined in Section 2.1.6 it is necessary to create such channels. For the vast majority of covert channels implementations are not publicly available or simply do not exist. Furthermore, communication over a covert channel does require more than just sending bits (modulation and demodulation). For example, a framing technique is needed to identify blocks of bytes in the bit stream (byte synchronisation) and an error correction technique is needed if the channel is noisy.

Instead of (re)implementing techniques as separate tools we chose to design and implement a modular framework where for example, a specific framing technique can be used with different covert channels. The design of our framework follows the layered approach traditionally used in the design of communication networks. A layer receives a data block from a lower/higher layer, removes/adds header and trailer, and passes the modified data block to the next higher/lower layer without knowing any details of other layers. However, cross-layer information exchange is also possible if really necessary. Another advantage of an integrated tool is that it can create a single channel that actually is a combination of multiple covert channels.

This appendix presents the design and implementation of the Covert Channels Evaluation Framework (CCHEF), pronounced *chef*, which can be used to create covert channels in network protocols. CCHEF was used to prototype and evaluate TTL covert channels (see Chapter 3), packet timing covert channels (see Chapter 4) and covert channels in first person shooter games (see Chapter 5). CCHEF was designed for (semi-)passive covert channels that use existing application traffic as cover. Hence, the active temperature-based covert channel (see Chapter 6) had to be implemented as separate tool.

First we describe the overall goals and main features. Then we present the architecture of CCHEF. Finally, we show how CCHEF is configured and provide a simple example. A more detailed manual for users or developers of covert channels is provided in [245]. The total size of CCHEF is roughly 20 000 total Source Lines of Code (SLOC) [246]. CCHEF has been made publicly available under the GNU license [247].

A.1 Goals and features

The main goal of CCHEF is to create covert channels for research purposes:

- Evaluate the capacity, stealth and robustness of covert channels and compare different channels.
- Evaluate and compare mechanisms to eliminate, limit the capacity or detect different covert channels.
- Create test traffic for existing intrusion detection software or firewalls (overt traffic with covert channels).

No attempts were made to hide the presence of the CCHEF application itself. The sender and receiver are normal user space applications and if root privileges are needed for the execution, they need to be started as root. This allowed us to focus on the actual covert channel techniques, avoids facilitating possible misuse, and improves the portability, since techniques to hide executables are often operating system dependent.

There are two different types of scenarios we wanted to support:

1. CCHEF should work in real networks with real overt traffic. This allows evaluating covert channels across real networks and is mandatory for testing of existing intrusion detection software or firewalls.
2. Usually testing with real traffic is restricted to controlled testbeds where it is almost impossible to generate a realistic traffic mix from a larger number of hosts. Therefore, CCHEF should also be capable of running on a single host emulating the use of covert channels with overt traffic from trace files.

The following paragraphs describe the main features of CCHEF. CCHEF supports both storage and timing channels as long as they are (semi-)passive. CCHEF supports covert channels in the IP protocol and in higher-layer protocols (e.g. TCP, HTTP). It is planned to extend CCHEF in the future to also support link-layer channels. CCHEF currently only works with IPv4. Support for IPv6 may be added in the future.

CCHEF is very flexible and extensible. It is possible to create new covert channel modulation modules without need to modify any code of the framework itself. Furthermore, it is possible to easily modify or add new code for encryption, authentication, framing and reliable transport. The source code of CCHEF was written with portability in mind, so that CCHEF can be used on different operating systems. However, the primary development platform was Linux and currently some functions only work on Linux.

CCHEF embeds covert channels into any existing application traffic. This means the cover traffic is real and does not look suspicious in any way. CCHEF cannot be used

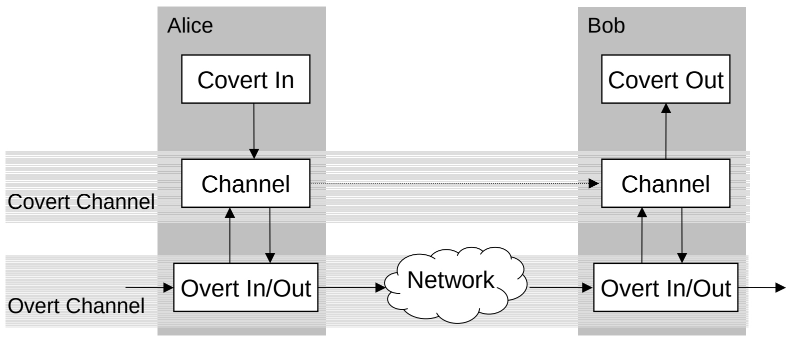


Figure A.1: CCHEF is used to transmit information across a network (Alice sending to Bob)

when the overt traffic needs to be very specific and tailored to the covert channel, such as required for temperature-based covert channels (see Chapter 6).

CCHEF is efficient in terms of CPU and memory usage, but the overall performance depends heavily on the implementation of the modules performing the modulation and providing reliable transport. In our experiments we confirmed that CCHEF can handle overt traffic of up to a few Mbit/s and trace files with up to a few hundred million packets.

A.2 Design and implementation

First we present an overview of CCHEF's architecture and the different layers. Then we discuss parts of the architecture in more detail: input/output devices, selection of overt packets, covert channel modulation modules and error simulation/emulation.

A.2.1 Usage modes

Figure A.1 shows how CCHEF is used for transmitting covert information across a network. For clarity the figure only shows the unidirectional channel of Alice sending to Bob, but channels in CCHEF can be bidirectional and therefore Bob could send to Alice at the same time.

Alice and Bob tap into an overt channel, which is a number of traffic flows between Alice and Bob. Alice and Bob can be the sender and receiver of the overt traffic instrumenting network applications or act as middlemen and use pre-existing overt traffic of unwitting users. Alice intercepts the overt traffic, encodes the covert data and re-injects the overt traffic with the embedded covert channel into the network. Bobs intercepts the overt traffic, decodes the covert data and possibly removes the covert channel.

Figure A.2 shows how CCHEF is used to evaluate covert channels based on overt traffic from trace files. In this case covert information is not actually sent across a network; Alice and Bob are a single entity. For each overt packet read from the trace file CCHEF first embeds the covert information, optionally simulates noise and then decodes

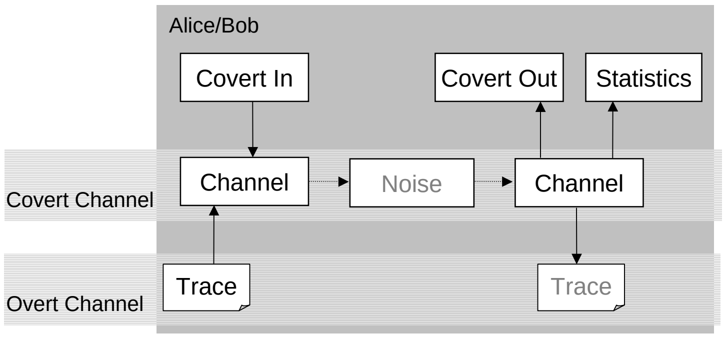


Figure A.2: CCHEF is used to evaluate covert channels based on overt traffic from trace files

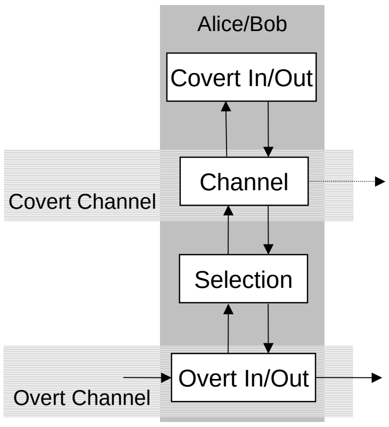


Figure A.3: CCHEF main modules

the covert information from the packet straight away. The received data is compared with the sent data and statistics are computed (e.g. error rate). The modified trace can be stored for later use, such as testing firewalls or intrusion detection systems.

A.2.2 Main modules

Figure A.3 shows the main modules of CCHEF. The heart of CCHEF is the *Channel* module that interfaces with multiple *Device* modules. Covert data to be sent is read from a *Covert In* device while received covert data is passed to a *Covert Out* device. The *Overt In/Out* device intercepts and re-injects overt traffic. The *Selection* module selects which packets of the overt traffic are used to encode the covert channel.

The Overt In/Out module intercepts overt packets in the send direction. A subset of the packets (carrier traffic) is chosen by the Selection module and passed on to the Channel module. Packets not selected are re-injected into the network. The Channel module encodes covert data into the carrier traffic and passes the modified packets back to the Overt In/Out device, which then re-injects them back into the network.

Overt packets arriving in the receive direction are intercepted by the Overt In/Out module and are passed on to the Channel module if selected by the Selection module.

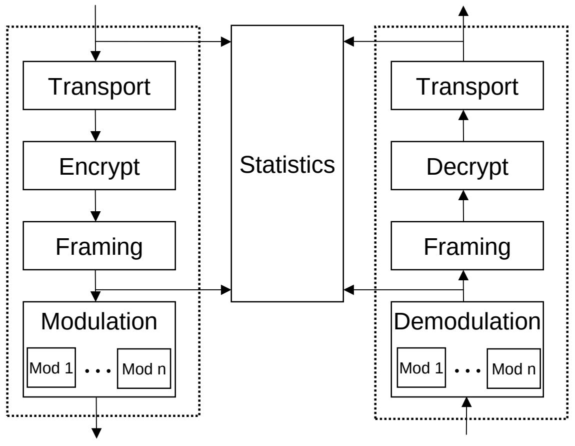


Figure A.4: CCHEF channel module functions at sender and receiver

The Channel module decodes covert data present in the packets and if possible removes the covert channel. CCHEF also supports passive receivers that work with copies of the original packets and do not delay the overt traffic, if removing the covert channel is not desired or not possible. Then, the Overt In/Out device intercepts copies of packets.

The Channel module performs all the functions necessary for covert communication such as modulation/demodulation of the covert data, segmenting the covert bits into blocks (framing/deframing), encryption/decryption and error detection and correction (see Section A.2.3). A configuration module configures all devices and the channel based on a configuration file (see Section A.3).

A.2.3 Channel module

The Channel module is composed of multiple sub modules, each representing a communication layer of the covert channel. Figure A.4 shows the details of the layers on the sender (left) and receiver (right). There are four main modules in CCHEF: modulation, framing, encryption and transport[1]. The modulation, framing and transport modules are similar to the physical, link and transport layers of the Open Systems Interconnect (OSI) model. CCHEF only supports communication between one covert sender and one or multiple covert receiver(s) and therefore the equivalent of the network layer capable of routing does not exist.

The *Modulation* modules are responsible for modulating/demodulating the covert bits into the overt packet stream and bit synchronisation (if needed). Each module implements a certain covert channel technique and provides modulation and demodulation functions. Multiple modulation modules can be used in parallel to encode covert data using different covert channel techniques simultaneously. However, Alice and Bob must use the same set

[1]Framing, Encryption or Transport modules can be null modules if their function is not required.

of modules with the same parameters in the same order. The implemented modules are described in Section A.2.6.

The *Framing* module is responsible for framing, including byte synchronisation. Each Framing module provides a framing and deframing method. The implemented modules are described in Section A.2.7. The *Transport* module is responsible for error detection and correction using FEC or ARQ techniques, and flow control (if needed). It also performs segmentation of the data into blocks of bytes. The implemented transport modules are described in Section A.2.8.

The *Encryption* module encrypts/decrypts the covert data. The main reason for having this type of module is not to actually secure the covert data, as CCHEF was developed for the sole purpose of research. But encryption can change the distribution of the input data, which may affect the characteristics of the channel. For example, the capacity of the binary symmetric channel is achieved only for uniformly random data [22]. Currently only a simple XOR-based algorithm is implemented.

During operation sender and receiver compute statistics, for example the total number of covert bits sent or received. They also log copies of the sent and received bit streams for later analysis, such as computing bit error rates. Alice logs the bit stream sent before transport and before modulation. Bob logs the bit stream received after demodulation and after transport. If CCHEF is used with overt traffic from traces the bit streams can be compared during operation. If a network separates Alice and Bob, the bit streams are stored and can be compared offline.

A.2.4 Device modules

Devices are used to input and output covert data, and to intercept and re-inject overt traffic. There are a number of different device modules. Table A.1 summarises the purposes of the different devices. A “yes” indicates the usual purpose of a device, whereas a “possible” indicates a possible but unlikely use.

The *Random* device generates a uniform random data stream (the probability of zeros and ones is equal). The *Null* device writes data to /dev/null, when the received data should be ignored. The *Text* device reads from a text file or outputs to a text file. The *Tunnel* device reads/writes IP packets from/to a tunnel network device. This enables CCHEF to tunnel IP packets across the covert channel. The *Netfilter Queue* device instruments the Netfilter queue framework inside the Linux kernel [248]. This framework consists of a number of hooks in the packet processing routines of the kernel. Packets are intercepted and delivered to a userspace application. The userspace application can modify the packets and re-inject them back into the kernel.

The *Libtrace* device reads/writes packets from/to various trace files and supports different formats, including libpcap format, Endace Record Format (ERF), and older formats

Table A.1: Device modules and their function

Device	Covert In	Covert Out	Overt In/Out
Random	yes	no	no
Null	no	yes	no
Text	yes	yes	no
Tunnel	yes	yes	possible
NetfilterQueue	possible	possible	yes
Pcap	possible	possible	yes
Libtrace	possible	possible	yes

used by DAG cards [197]. Its main purpose is to read overt traffic from trace files and to create trace files with embedded covert channels. However, it could also be used to send recorded IP packets across the covert channel. It is based on the libtrace library [249].

A.2.5 Packet selection and Selector modules

The packet selection module determines which overt packets are used as carrier for the covert channel. The module is composed of a number of sub modules: classifier module, flow-grouping module, and selector module (see Figure A.5).

Overt packets are usually obtained either via the Netfilter Queue device (real network) or via the Libtrace device (traffic traces). The *Classifier* module only passes on packets that match a set of configurable rules, based on the standard 5-tuple of source/destination IP address, protocol, and UDP/TCP ports. Non-matching packets are re-injected into the network immediately. The module limits the covert channel to overt traffic between the specified source and destination pair(s).

The *Flow Grouping* module groups packets into bidirectional flows according to the 5-tuple. Each flow is identified by a ‘unique’ flow ID. Packets are grouped into flows because many covert channel techniques encode data relative to flows and need to keep per-flow state (e.g. some TTL covert channels, packet timing channels, game traffic channel). Flows are ended by a timeout or by a TCP connection teardown[2], in which case flow state of modulation modules is deleted (see below). The first packet of a flow defines the ‘forward’ direction. Packets with IP addresses and ports reversed are recognised as going in the ‘backward’ direction.

Finally, packets are passed to the *Selector* module. The purpose of this module is to select only specific packets within a flow or across flows as carrier for the covert data, possibly determined by a secret shared between Alice and Bob. Packets meeting the selection criteria are passed on to the modulation module. Modified packets are re-injected into the network by the Overt In/Out device.

[2]CCHEF tracks TCP state but does not implement a full TCP state machine for performance reasons.

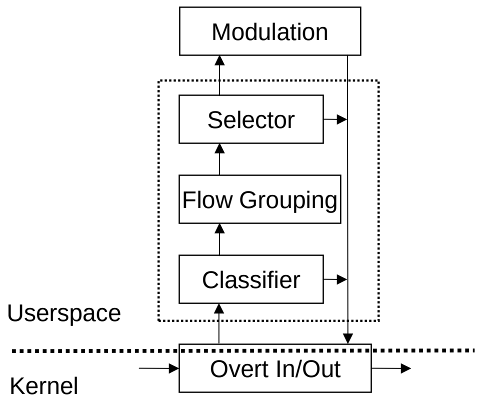


Figure A.5: Selection of overt packets to be used as carrier for covert data

Currently, the following Selector modules exist. The *All* module selects every packet. The *Hash* module selects a fraction of the overt traffic based on sparse encoding (see Section 4.2.2). The *TTLHash* module implements modified sparse encoding for the TTL channel (see Section 7.2.2).

A.2.6 Modulation modules

Modulation modules are responsible for encoding covert data into the overt traffic and for decoding (and possibly removing) covert data from the overt traffic. Modulation modules are realised as shared libraries. This means new modules can be added without the need to modify or recompile CCHEF.

A module has global constructor and destructor functions to initialise and destroy global state. A module also has per-flow initialise and destroy functions, which are called each time a new flow starts or an existing flow ends (as determined by CCHEF's flow-grouping module). The per-flow initialise function allows a module to allocate per-flow state. For each selected overt packet CCHEF calls either the encode or the decode function of a module depending on the direction of the overt packet as determined by the Classifier (forward or backward). CCHEF passes pointers to the packet data, packet meta-data (such as packet arrival timestamps) and the allocated per-flow state (if any) to the encode and decode functions.

CCHEF also provides timers for modulation modules. A module can register multiple timers at start-up and every time a new flow starts. Each timer is characterised by a unique ID and a timeout value. When a timer expires CCHEF calls the timeout function of the module. Each time the timeout function is called the module can adjust the timeout value of the expired timer, which includes destroying the timer.

Figure A.6 summarises the various functions of modulation modules called by the Channel module. We implemented a number of modulation modules. As simple example, we implemented the covert channel in the IP ID field as described in [36]. Several

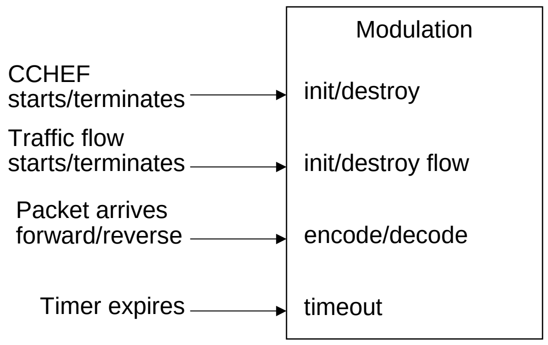


Figure A.6: Modulation module functions

modules were developed for different techniques to encode covert information in the TTL field (see Chapter 3). Two modules were developed for encoding covert information into the timing of packets (see Chapter 4). The game traffic covert channel described in Chapter 5 was implemented as another modulation module.

A.2.7 Framing modules

Framing modules are mainly responsible for framing, but can also provide other features, such as error detection and correction. The sender segments the data into fixed-size or variable-size blocks of bytes. We call these blocks frames because we consider this process as being similar to the link layer in the OSI model. The sender must segment the data such that the receiver can identify frame boundaries in the incoming bit stream.

Framing modules are realised as C++ classes. Currently six modules exist. The *Start Of Frame* (SOF) module implements the framing technique used by Ethernet (HDLC). The *CRC* framing module identifies frames based on CRC32 (as done by ATM). The *Null* module does nothing and is useful for analysis of channels without using a framing technique. The *RS* framer identifies frames based on RS codes and the *SOF2* framer is used for reliable transport with packet loss (see Section 3.4). There is also a specific framing module for the game traffic covert channel (see Section 5.4).

A.2.8 Transport modules

Transport modules segment data into blocks and provide functions to make the communication over noisy covert channels reliable. For example, they add sequence numbers or perform FEC so that lost frames can be detected and corrupt frames can be corrected. They can also implement ARQ schemes in case of bidirectional channels.

Transport modules are realised as C++ classes. Currently there are three transport modules. The *Simple* module provides 8-bit sequence numbers for detecting lost data blocks and FEC based on RS codes. The *Marker* transport module implements a scheme for reliable data transport over channels with deletions using a combination of marker codes and RS codes (see Section 3.4). The *Null* module does nothing and can be used if the underlying channel is error-free.

A.2.9 Error simulation modules

CCHEF can simulate channel errors. This is most useful in trace-file mode (see Figure A.2). The error simulation is done at the sender after the bit modulation. An error module has two functions, which are called with a pointer to the current overt packet and its meta-data. The *pre* function is called prior to modulation, allowing the error module to learn the original unmodified packet data. The *post* function is called after the modulation and enables the module to modify the overt packet with embedded covert data and thus to simulate various error characteristics of the covert channel.

The error simulation was designed to simulate channel errors caused by the modification of data fields in the packet (storage channel) or timing noise (timing channel) on the path between Alice and Bob. However, lost or reordered IP packets also introduce errors on the covert channel. CCHEF supports the emulation of overt packet loss and reordering, but currently only uniform random distributions are implemented.

A.3 Configuration

This section illustrates how CCHEF is configured. For further information the reader is referred to [245]. An XML configuration file controls the behaviour of CCHEF. The file is divided into several parts. In each part the configuration information is specified as preferences (PREFs). Preferences have a name, a value and (optionally) a type specification [245]. Figure A.7 shows an example configuration file.

The `MAIN` section defines general settings e.g. the name of the log file. `MODULE` specifications define modulation modules, `DEVICE` specifications define input and output devices, `FRAMER` specifications define framing techniques, and `TRANSPORT` specifications define transport protocols. Optional specifications define encryption techniques, packet selection techniques, and noise simulation methods. Finally, the covert channel is defined by specifying the devices for the input and output of covert data, the source of overt packets used as cover, the covert channel module(s) and the framing and transport techniques, as well as optionally packet selection, encryption and noise simulation methods.

A channel must specify `Cover` and either `CovertIn` or `CovertOut` (unidirectional channel) or both (bidirectional channel) referring to devices specified. It must also specify one or more modules under `Modules` referring to module(s) specified. If multiple

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CONFIG SYSTEM "config.dtd">
<CONFIG>
  <MAIN>
    <!-- general settings -->
  </MAIN>
  <MODULE NAME="ttl">
    <PREF NAME="BitsPerPacket">1</PREF>
    <PREF NAME="Delta">1</PREF>
  </MODULE>
  <TRANSPORT NAME="Simple">

    <PREF NAME="Type">Simple</PREF>
    <PREF NAME="BlockSize">16</PREF>
    <PREF NAME="Parity">0</PREF>
  </TRANSPORT>
  <FRAMER NAME="SOF">
    <PREF NAME="Type">SOF</PREF>
  </FRAMER>
  <DEV NAME="InFile">
    <PREF NAME="Type">File</PREF>
    <PREF NAME="Filename">send.txt</PREF>
  </DEV>
  <DEV NAME="OutFile">
    <PREF NAME="Type">File</PREF>
    <PREF NAME="Filename">recv.txt</PREF>
  </DEV>
  <DEV NAME="NFQueue">
    <PREF NAME="Type">NetfilterQueue</PREF>
    <PREF NAME="Filter">src_host 192.168.1.123 AND dst_host 192.168.4.45</PREF>
  </DEV>
  <CHANNEL NAME="channel1">
    <PREF NAME="CovertIn">InFile</PREF>
    <PREF NAME="CovertOut">OutFile</PREF>
    <PREF NAME="Cover">NFQueue</PREF>
    <PREF NAME="Framer">SOF</PREF>
    <PREF NAME="Transport">Simple</PREF>
    <PREF NAME="Modules">ttl</PREF>
  </CHANNEL>

</CONFIG>
```

Figure A.7: Example CCHEF configuration file

modules are used, sender and receiver configuration files must list the modules in exactly the same order.

Appendix B
Time-to-live Covert Channels

This appendix contains additional material for TTL covert channels analysed in Chapter 3 and Chapter 7.

B.1 TTL error probability distributions

Figures B.1 and B.2 show the TTL error distributions for the Grangenet, Twente, Waikato and Bell traces, which are not shown in Section 3.1.5.

B.2 Error probability analysis

In this section we derive error probabilities for the different modulation schemes described in Section 3.2. Let the discrete random variable X_i be the TTL error of a packet i and X the error probability distribution over all packets. We base our analysis on the following assumptions:

1. The covert data is uniformly random distributed (the probability of a zero or one being transmitted is equal to $\frac{1}{2}$). This is typically the case if the data is encrypted.

2. We assume that one bit of covert data is encoded per TTL, since otherwise the stealth would be severely reduced (see Section 3.1).

3. We assume all X_i are independent identical distributed (iid) random variables and the probability distribution is stationary.

For direct encoding techniques the error probability only depends on the error occurring for each packet independently of other packets. An error occurs if the absolute value of the TTL error is greater than zero and an odd number. Because even errors do not modify the lowest bit, they do not cause errors on the covert channel. Since the maximum TTL value is 255 the error probability is:

$$P_{\mathrm{D}} = \sum_{k=-128}^{127} P(X = 2k+1) \; . \tag{B.1}$$

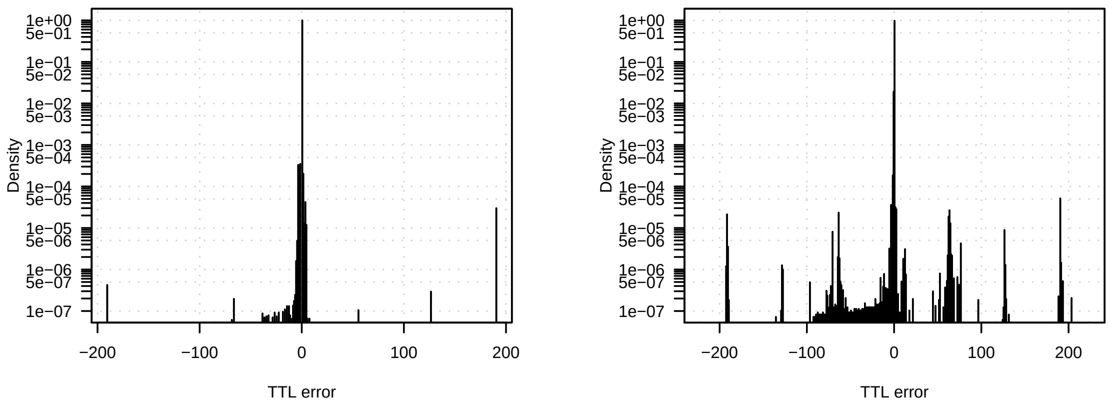


Figure B.1: TTL error distribution for the Grangenet trace (left) and Twente trace (right)

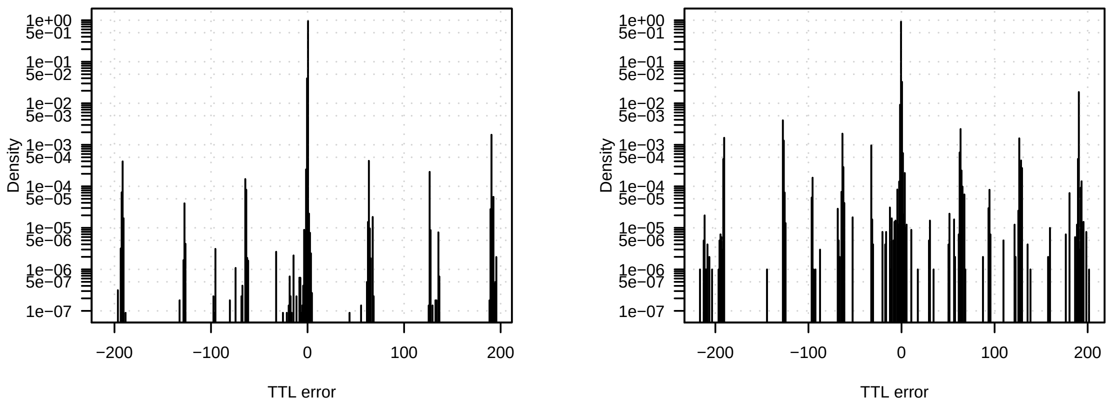


Figure B.2: TTL error distribution for the Waikato trace (left) and Bell trace (right)

Mapped encoding schemes encode a logical zero and a logical one as two different TTL values. Usually one of the TTL values is the most common TTL value of a packet flow and the other value is a slight modification.

We assume that the receiver either knows the mapping or learns it by watching the TTL sequence and assuming the two most common TTL values are the symbols for a logical zero and a logical one. Then the error probability only depends on the error occurring for each packet independently of other packets.

First we derive the error probability for MED encoding. The error probability for $0 \rightarrow 1$ and $1 \rightarrow 0$ errors is not identical. The probability for $0 \rightarrow 1$ errors is:

$$P_{0\rightarrow 1} = P(X \leq -\left\lceil \frac{\Delta}{2} \right\rceil) , \tag{B.2}$$

where $\lceil . \rceil$ is the ceiling function. The probability for $1 \to 0$ errors is smaller for even Δ because we assume the receiver decodes a logical one in case the received symbol is exactly the threshold value (the middle between a logical zero and logical one):

$$P_{1\to 0} = P(X \geq \left\lceil \frac{\Delta}{2} + \frac{1}{2} \right\rceil) \,. \tag{B.3}$$

Given the first assumption, how the receiver decides at the threshold does not affect the overall error probability. However, in general it is best to decode the threshold value as the bit occurring most frequently in the data. The overall error probability follows from Equations B.2 and B.3:

$$\begin{aligned} P_{\text{MED}} &= \frac{P_{0\to 1}(\Delta)}{2} + \frac{P_{1\to 0}(\Delta)}{2} \\ &= \frac{P(X \leq -\left\lceil \frac{\Delta}{2} \right\rceil)}{2} + \frac{P(X \geq \left\lceil \frac{\Delta}{2} + \frac{1}{2} \right\rceil)}{2} \,. \end{aligned} \tag{B.4}$$

In the same way the error probability for MEI is derived to:

$$P_{\text{MEI}} = \frac{P(X \geq \left\lceil \frac{\Delta}{2} \right\rceil)}{2} + \frac{P(X \leq -\left\lceil \frac{\Delta}{2} + \frac{1}{2} \right\rceil)}{2} \,. \tag{B.5}$$

If the error distribution is symmetric P_{MEI} and P_{MED} are identical.

Differential encoding schemes encode covert bits as change between two TTL values and therefore the error probability depends on the difference of the two errors. Let $Z = Y - X$ be the difference of the two error distributions of two consecutive packets x and y. Then the probability that Z is larger than some integer z can be computed using the discrete convolution [250]:

$$P(Z \geq z) = \sum_{m=z}^{\infty} \sum_{n=-\infty}^{\infty} P(X = n) \cdot P(Y = m + n) \,. \tag{B.6}$$

For AMI encoding a $0 \to 1$ error occurs when the absolute value of Z is larger than $\frac{\Delta}{2}$ (assuming at the threshold the receiver decodes a logical zero). A $1 \to 0$ error occurs when Z is in the interval $[\frac{\Delta}{2}, \frac{3\Delta}{2} + \frac{1}{2})$ and the bit is encoded as TTL decrease or when Z is in the interval $(-\frac{3\Delta}{2} - \frac{1}{2}, -\frac{\Delta}{2}]$ and the bit is encoded as TTL increase. This is because any TTL change larger than $\frac{\Delta}{2}$ is decoded as logical one. The probability that a logical one is encoded as increase or decrease is $\frac{1}{2}$ given the first assumption. Then the overall error probability is:

$$
\begin{aligned}
P_{\text{AMI}} &= \frac{P_{0\to1}(\Delta)}{2} + \frac{P_{1\to0}(\Delta)}{2} \\
&= \frac{P(|Z| > \frac{\Delta}{2})}{2} + \frac{\frac{1}{2}P(\lceil\frac{\Delta}{2}\rceil \le Z < \lceil\frac{3\Delta}{2} + \frac{1}{2}\rceil)}{2} \\
&+ \frac{\frac{1}{2}P(-\lceil\frac{3\Delta}{2} + \frac{1}{2}\rceil < Z \le -\lceil\frac{\Delta}{2}\rceil)}{2} \; .
\end{aligned} \qquad \text{(B.7)}
$$

For DUB a $0 \to 1$ error occurs when Z is larger than Δ (assuming at the threshold the receiver decodes a logical zero) and a $1 \to 0$ error occurs when Z is smaller or equal than $-\Delta$. The overall error probability is:

$$
\begin{aligned}
P_{\text{DUB}} &= \frac{P_{0\to1}(\Delta)}{2} + \frac{P_{1\to0}(\Delta)}{2} \\
&= \frac{P(Z > \Delta)}{2} + \frac{P(Z \le -\Delta)}{2} \; .
\end{aligned} \qquad \text{(B.8)}
$$

We simulated all modulation schemes and measured the error rates. CCHEF was used to simulate a channel between covert sender and receiver. The sender-part of the simulator encoded covert bits into the TTL fields of overt packets. Then the packets' TTL values were modified to simulate the channel error. Finally, the receiver-part of the simulator decoded the covert bits from the overt packets. Finally, we computed the error rate, which is the number of wrongly decoded bits divided by the total number of bits.

We used a simple idealised TTL noise model. The error was simulated using a Normal distribution with a mean of zero and standard deviations $\sigma = \{1.0, 1.5\}$. The values of σ were chosen such that the resulting error rates were similar to the error rates of empirical TTL error distributions. In practice the actual error probabilities can be computed based on the empirical error distributions (see Section 3.1).

In all simulations we used uniform random covert data. The overt data was a synthetic packet trace with approximately 42 million packets. The initial TTL value was always set to 128. Let A be the peak-to-peak signal amplitude of the encoding schemes, the difference between the signal level of logical one and zero. Then for direct schemes $A = 1$, for DUB $A = 2\Delta$ and for all other techniques $A = \Delta$. We varied the amplitude within a limited range ($1 \le A \le 6$). Every experiment was repeated 20 times.

Since the Normal distribution is symmetric both mapped error probabilities (Equations B.4 and B.5) give identical results. Therefore we only simulated MED as representative for both techniques. For direct encoding schemes we assumed knowledge of the true hop count at the receiver.

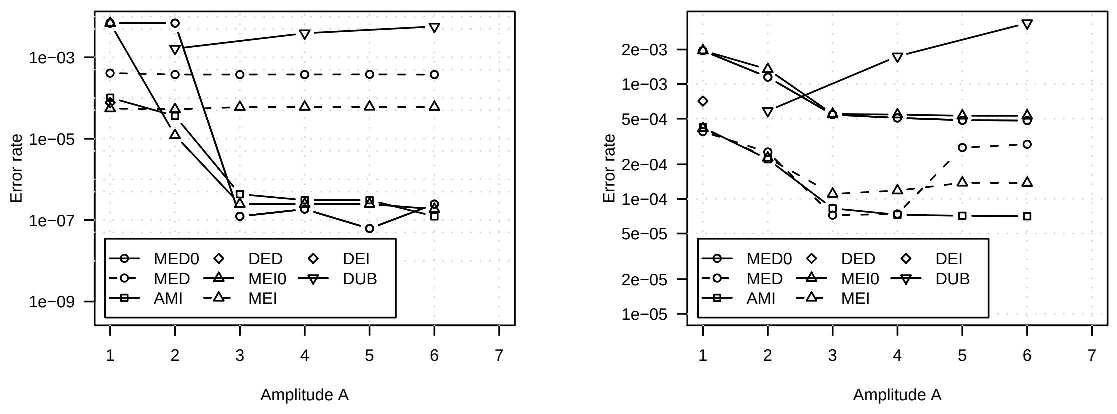


Figure B.3: Error rate for different modulation schemes and amplitudes for Grangenet trace (left) and Twente trace (right)

For comparing the simulation results with the theoretical probabilities we used the relative root mean square error (RMSE), which is the RMSE of the simulated error rates x_i compared to the theoretical error probability $\hat{x}$ divided by $\hat{x}$:

$$\delta\text{RMSE} = \frac{\text{RMSE}}{\hat{x}} = \frac{\sqrt{\frac{1}{N}\sum_i(\hat{x}-x_i)^2}}{\hat{x}} . \qquad \text{(B.9)}$$

The results show that the relative RMSE is always less than 2% indicating a good match between the theoretical error probabilities and the simulation results.

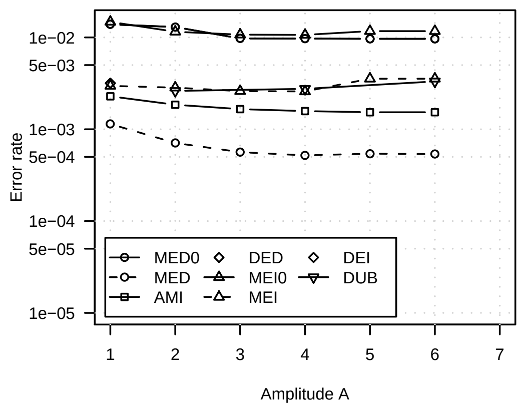
B.3 Error rate for modulation schemes and traces

Figures B.3 and B.4 show the error rates of the different modulation schemes depending on the amplitude for the Grangenet, Twente, Waikato and Bell traces, which are not shown in Section 3.5.2.

B.4 Hop count differences instead of TTL differences

We investigated if the error rate for mapped and differential schemes is reduced by using hop count differences instead of TTL differences. Note, that this technique does not work if modulated TTL values cross boundaries between different initial TTL regions (e.g. increasing a TTL value of 62 by 4 would change the estimated hop count from 2 to 62). This is the case for MEI and DUB even for very small amplitudes, because they increase TTL values.

Figure B.5 shows the average error rates for MED0, MED and AMI when using hop count differences on the same scale as Figure 3.13. Error rates with TTL differences



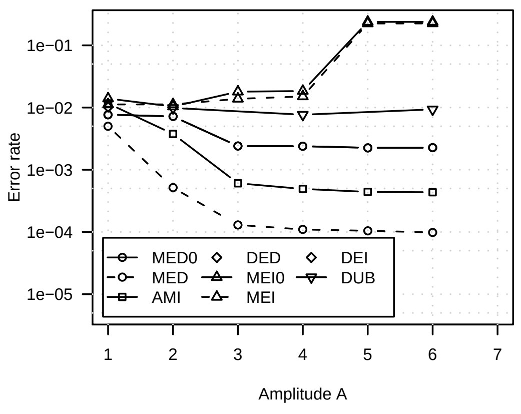


Figure B.4: Error rate for different modulation schemes and amplitudes for the Waikato trace (left) and Bell trace (right)

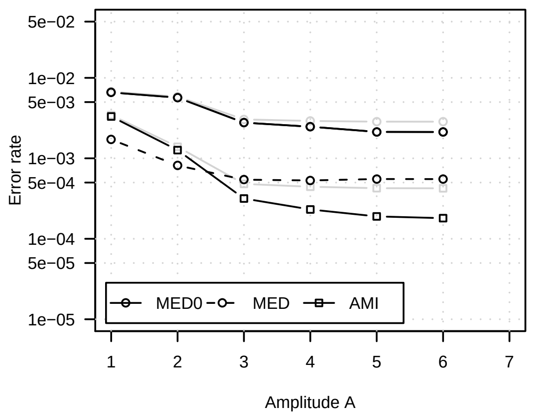


Figure B.5: Average error rate of MED0, MED and AMI encoding schemes across all traces when using hop-count differences

are shown as gray curves. MED0 and AMI show some notable improvements across all datasets for larger amplitudes. However, for MED the improvement is negligible.

B.5 Error rate changes over time

Figures B.6 and B.7 show the error rate changes over time for the MED modulation scheme and the Grangenet, Twente, Waikato and Bell traces, which are not shown in Section 3.5.4.

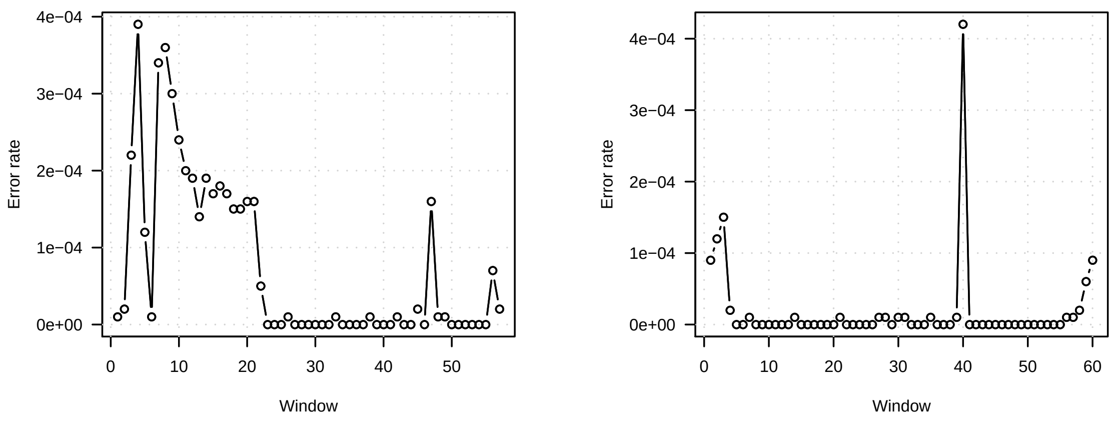


Figure B.6: Error rate for the MED modulation scheme over consecutive windows of 100 000 bits for Grangenet (left) and CAIA (right)

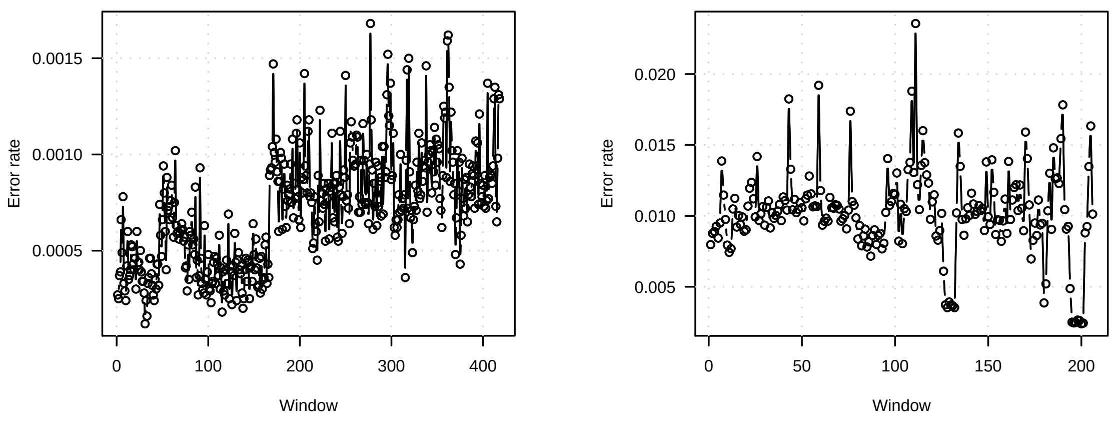


Figure B.7: Error rate for the MED modulation scheme over consecutive windows of 100 000 bits for Twente (left) and Bell (right)

B.6 Number of retransmissions for ARQ scheme

Here we derive the average number of (re)transmissions needed in a selective repeat ARQ scheme, given the actual bit substitution and deletion rates and a target block corruption probability. The average number of (re)transmission is used to compute the throughput of a combined FEC+ARQ scheme in Section 3.5.5, Section 3.5.6 and Section 4.5.4.

Given substitution and deletion probabilities p_S and p_D the corruption probability for a data block is:

$$p_B = 1 - (1 - p_E)^N \ , \tag{B.10}$$

where $p_E = p_S + p_D$ and N is the size of the block in bits including sequence numbers and checksums. The probability that a block is corrupted after T (re)transmissions is:

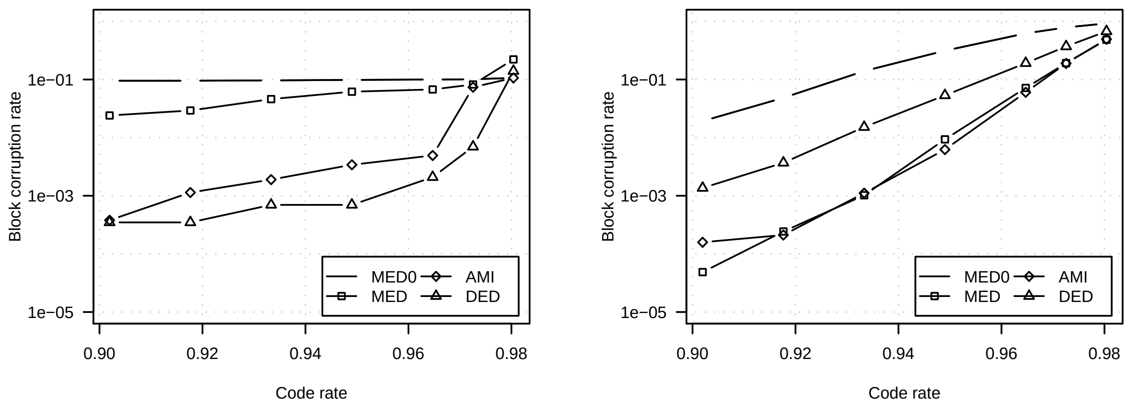


Figure B.8: Block corruption rate depending on code rate for the Grangenet trace (left) and Twente trace (right) without packet reordering and loss

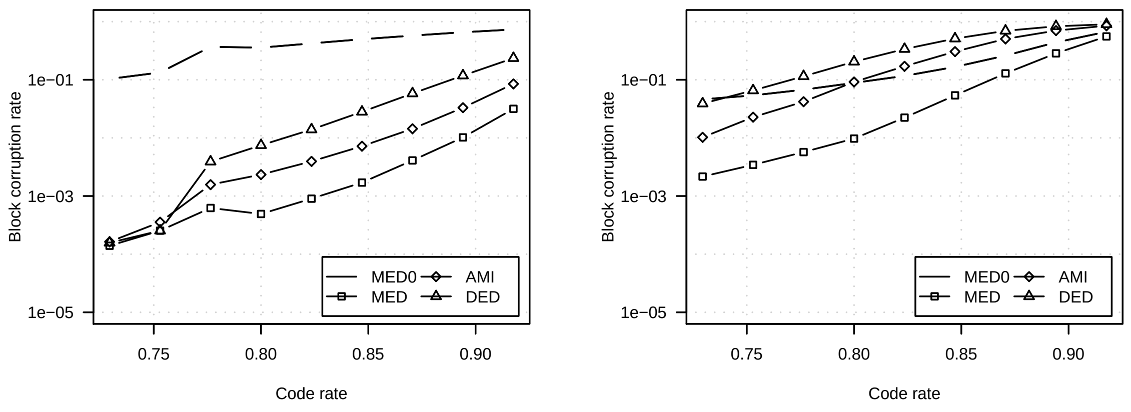
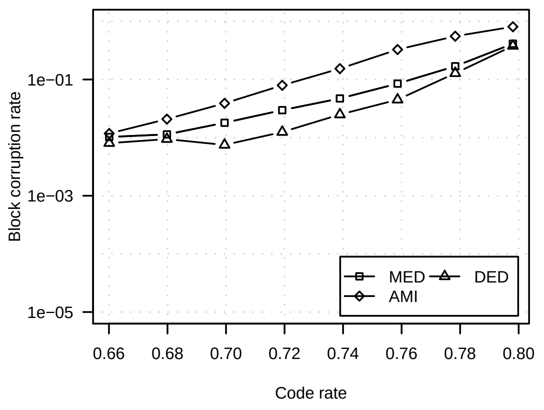


Figure B.9: Block corruption rate depending on code rate for the Waikato trace (left) and Bell trace (right) without packet reordering and loss

$$\hat{p}_{\mathrm{B}} = p_{\mathrm{B}}^{T} . \tag{B.11}$$

From equation B.11 follows that given the actual block corruption probability and a target block corruption probability $\hat{p}_{\mathrm{B}}$ the average number of (re)transmissions needed is:

$$\overline{T}\left(p_{\mathrm{B}}, \hat{p}_{\mathrm{B}}\right) = \max\left(1, \frac{\log\left(\hat{p}_{\mathrm{B}}\right)}{\log\left(p_{\mathrm{B}}\left(p_{\mathrm{E}}, N\right)\right)}\right) . \tag{B.12}$$



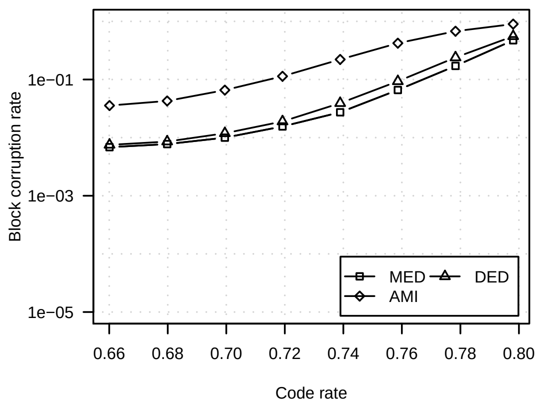


Figure B.10: Block corruption rate depending on code rate for Waikato (left) and Bell (right) for 0.5% packet loss

B.7 Block corruption rate

The following graphs show the block corruption rate depending on the code rate for the traces not shown in Section 3.5.5, but only for experiments where graphs are shown for the CAIA and Leipzig traces in Section 3.5.5.

Figure B.8 and B.9 show the block corruption rate over the code rate for the Grangenet, Twente, Waikato and Bell traces and the different modulation schemes without packet reordering and loss. Figures B.10 and B.11 show the block corruption rate depending on the code rate for the Grangenet, Twente, Waikato and Bell traces and the different modulation schemes for 0.5% packet loss without packet reordering. Figures B.12 and B.13 show the block corruption rate depending on the code rate for the Grangenet, Twente, Waikato and Bell traces and the different modulation schemes for 0.1% packet loss and 0.5% packet reordering.

B.8 Throughput for trace files

This section shows the graphs of throughput depending on TTL error rate, packet loss and reordering for the traces not shown in Section 3.5.5.

Figures B.14 and B.15 show the throughput depending on the packet loss rate for the Grangenet, Twente, Waikato and Bell traces with 0% packet reordering. Figures B.16 and B.17 show the throughput depending on the packet reordering rate for the Grangenet, Twente, Waikato and Bell traces with 0.1% packet loss rate.

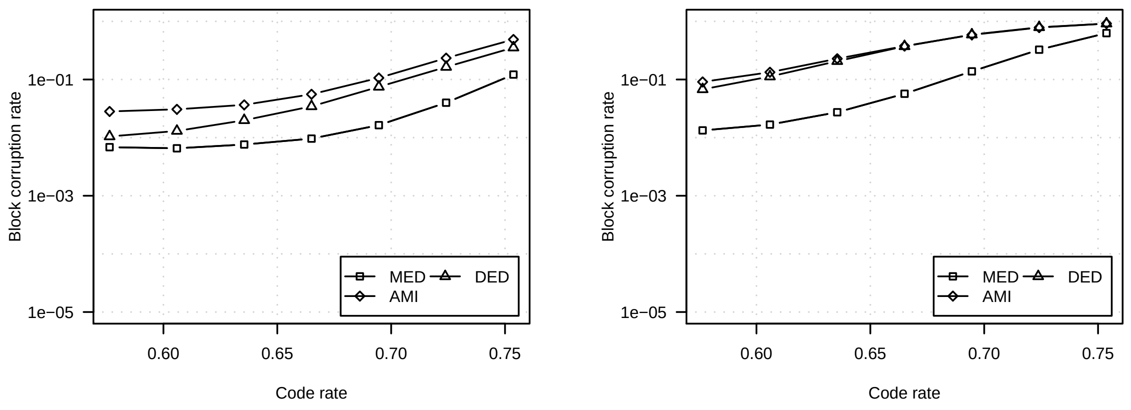


Figure B.11: Block corruption rate depending on code rate for Waikato (left) and Bell (right) for 0.5% packet loss

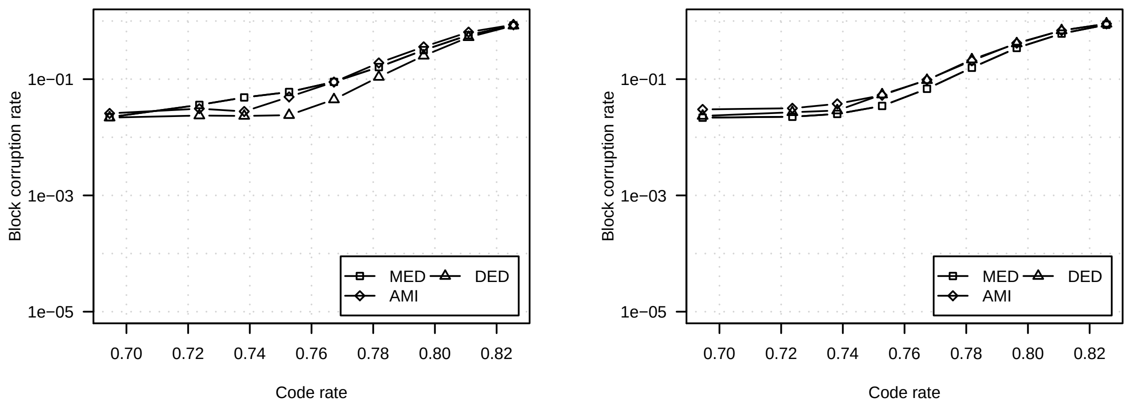


Figure B.12: Block corruption rate depending on code rate for Grangenet (left) and Twente (right) for 0.1% packet loss and 0.5% packet reordering

B.9 Decoding speed

Tables B.1 and B.2 show the average decoding speed for the reliable transport techniques proposed in Section 3.4 for the best modulation scheme at the optimum code rate and compares it to the average throughput of the best modulation scheme for the different traces and packet loss rates. The performance was measured on a PC with Intel Core2 2.4 GHz CPU and 4 GB of memory. The results show that even for the experiments with the highest error rate the decoder is still much faster than the actual throughput of the channel.

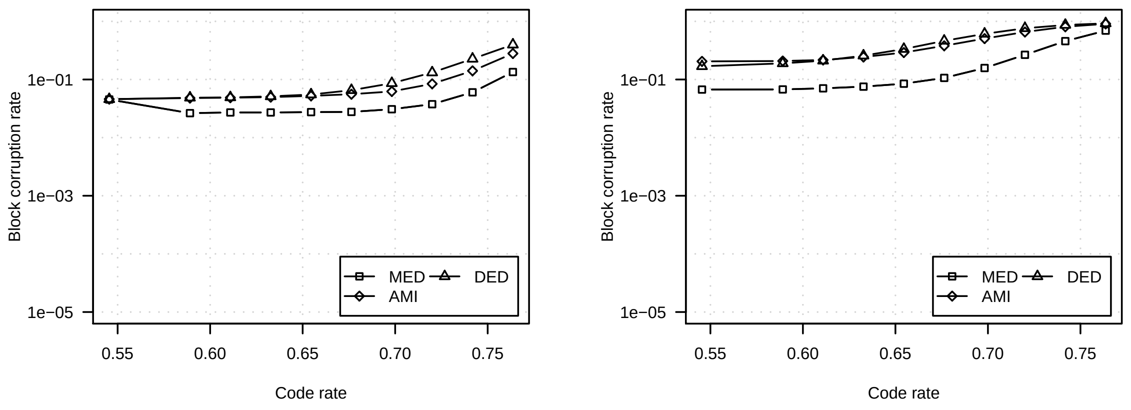


Figure B.13: Block corruption rate depending on code rate for Waikato (left) and Bell (right) for 0.1% packet loss and 0.5% packet reordering

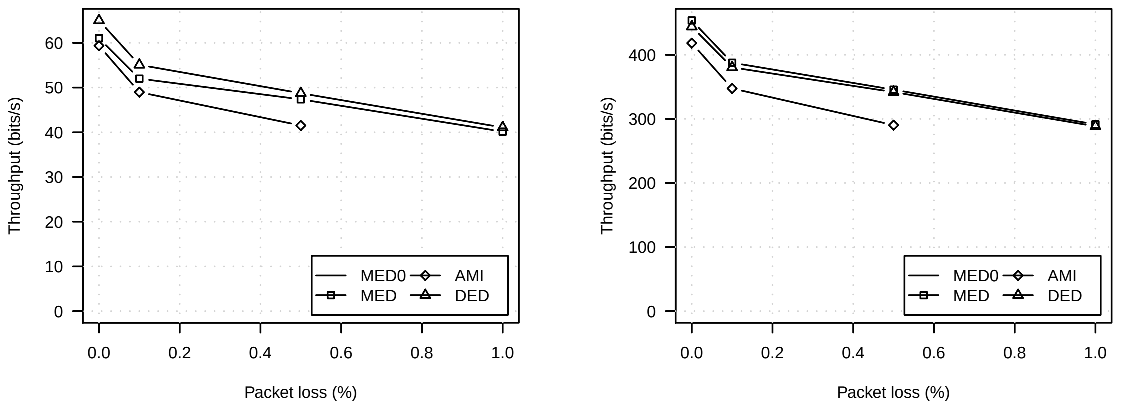


Figure B.14: Throughput depending on the packet loss rate for Grangenet (left) and Twente (right) for 0% packet reordering

B.10 Covert bit rate over time

Here we analyse the average bit rate of the TTL channel over time. We analyse the raw bit rate after demodulation, but prior to framing and error correction. We use the data from the experiments described in Section 3.5.6. Figure B.18 plots the bit rate over time for Q3 client-to-server traffic and scp traffic (direction of the data), averaged over non-overlapping windows of 500 bits, for the MED modulation scheme with 0% and 1% packet loss and emulated Pareto-distributed jitter with $\sigma = 0.2$ ms.

The results show that for Q3 client-to-server traffic the bit rate is constant even for 1% packet loss for both encoding schemes, apart from the initial connection phase and map changes occurring every 600 seconds. Packet reordering has no effect on the burstiness. For Q3 server-to-client and SSH traffic the results are very similar and hence not shown.

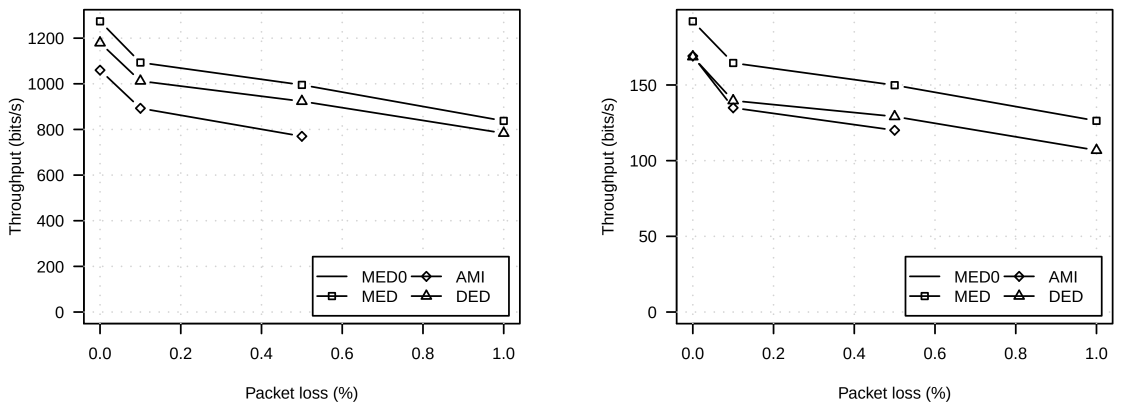


Figure B.15: Throughput depending on the packet loss rate for Waikato (left) and Bell (right) for 0% packet reordering

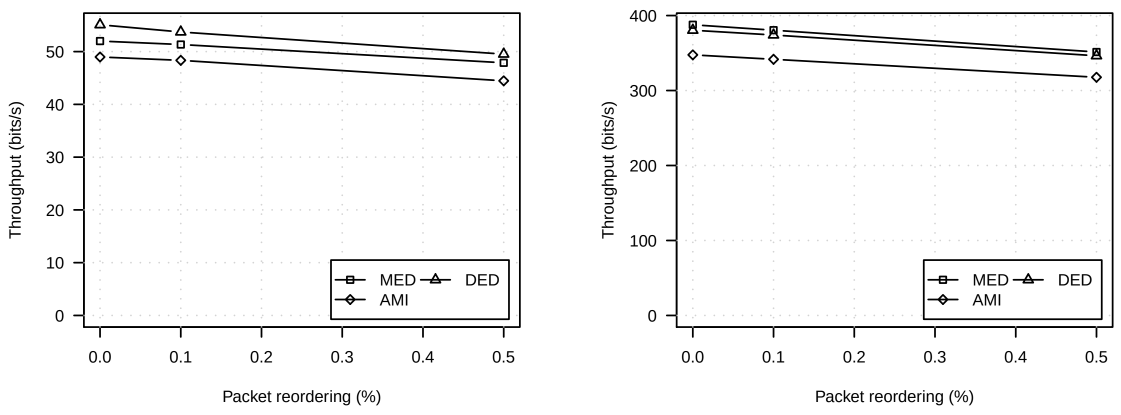


Figure B.16: Throughput depending on the packet reordering rate for Grangenet (left) and Twente (right) with 0.1% packet loss

For scp traffic the bit rate is almost constant without packet loss. With packet loss the overt packet rate and hence the covert bit rate fluctuates greatly. The higher the loss rate is, the greater the burstiness is. We leave a more comprehensive study as future work.

B.11 Code parameters

Here we describe the various codes used in the trace file and testbed experiments described in Section 3.5.5 and 3.5.6.

In the trace file experiments we separated the trace files into two groups where each group contains traces with similar TTL error rates. CAIA, Grangenet and Twente were in the *low error* rate group and Bell, Leipzig, Waikato were in the *high error* rate group. In the experiments we varied the redundancy of the RS coder (K) separately for each group.

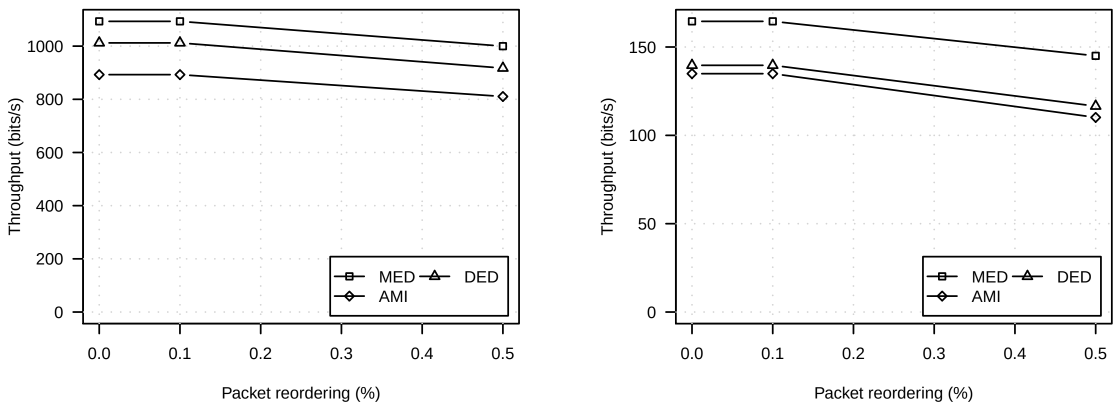


Figure B.17: Throughput depending on the packet reordering rate for Waikato (left) and Bell (right) with 0.1% packet loss

Table B.1: Average throughput and receiver decoding speed for best encoding scheme depending on packet loss rate (0% reordering)

		Loss 0%		**Loss 0.1%**	
Trace	**Scheme**	**Throughput (bits/s)**	**Decoding Speed (bits/s)**	**Throughput (bits/s)**	**Decoding Speed (bits/s)**
CAIA	MED	69	17.9M	58	6.5M
Grangenet	DED	65	17.0M	55	6.0M
Twente	MED	454	14.5M	388	5.4M
Waikato	MED	1 273	8.1M	1 093	4.8M
Bell	MED	192	10.2M	165	3.6M
Leipzig	MED	10.6k	11.6M	9 102	4.8M

Table B.3 shows the used parameters depending on the trace group and packet loss and reordering rate. For the outer marker code we state the size of the preamble, whereas for the inner marker code we state the number of bits between markers followed by the number of marker bits. For the RS code we state *N*,*K*. Note that *K* includes the CRC32 checksum and 8-bit sequence number part of each block. The last column shows the overall block size in bytes.

Table B.4 shows the code parameters used in the different testbed experiments in the same format. TTL errors and packet loss caused identical error rates for all applications. For packet reordering we tuned the emulated delay so that error rates were similar for the different applications. Hence we used the same codes for all applications. The table also shows the code rate, which was verified from the experimental data.

Table B.2: Average throughput and receiver decoding speed for best encoding scheme depending on packet loss rate (0% reordering)

		Loss 0.5%		**Loss 1.0%**	
Trace	**Scheme**	**Throughput (bits/s)**	**Decoding Speed (bits/s)**	**Throughput (bits/s)**	**Decoding Speed (bits/s)**
CAIA	MED	51	3.2M	43	2.7M
Grangenet	DED	49	3.2M	41	2.6M
Twente	MED	346	3.4M	291	2.4M
Waikato	MED	995	3.8M	838	2.3M
Bell	MED	150	2.5M	126	1.4M
Leipzig	MED	8 267	3.5M	6 934	2.0M

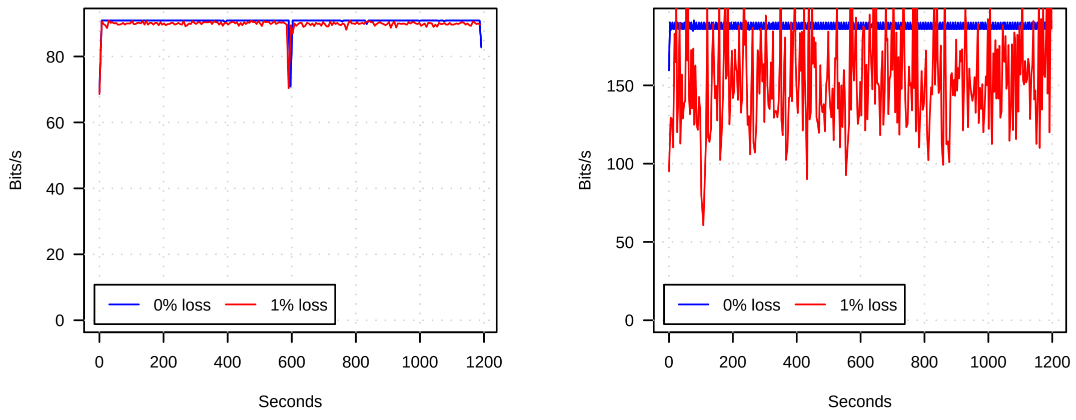


Figure B.18: Average bits per second over time for Q3 client-to-server traffic (left) and scp traffic (right)

B.12 Detection results

This section contains the detection accuracy and classifier complexity results for the DED modulation scheme. The results for the MED modulation scheme are shown in Section 7.2. Figure B.19 shows the flow-based and bit-based detection accuracy for both traces using sparse encoding and modified sparse encoding. Figure B.20 shows the overall detection accuracy depending on the different feature sets, described in Section 7.2, for both traces using modified sparse encoding. Figure B.21 shows the complexity of the classifier depending on the feature sets.

Table B.3: Code parameters (trace file experiments)

Trace group (TTL error)	Loss (%)	Reordering (%)	Outer Marker	Inner Marker	RS	Block size (bytes)
Low	0	0	NA	NA	255,255–235	255
Low	0.1	0	40 bit	16,2	240,236–216	275
Low	0.1	0.1	40 bit	16,2	240,236–208	275
Low	0.1	0.5	40 bit	16,2	240,232–196	275
Low	0.5	0	40 bit	16,2	176,167–139	203
Low	1.0	0	40 bit	16,3	176,155–131	214
High	0	0	NA	NA	255,239–191	255
High	0.1	0	40 bit	16,2	240,227–179	275
High	0.1	0.1	40 bit	16,2	240,227–179	275
High	0.1	0.5	40 bit	16,2	240,215–155	275
High	0.5	0	40 bit	16,2	176,158–122	203
High	1.0	0	40 bit	16,3	176,146–110	214

Table B.4: Code parameters (testbed experiments)

TTL error (%)	Loss (%)	Reordering (%)	Outer Marker	Inner Marker	RS	Block size (bytes)	Code rate
0	0	0	NA	NA	73,69	73	0.88
0.001	0	0	NA	NA	73,69	73	0.88
0.1	0	0	NA	NA	77,69	77	0.83
1.0	0	0	NA	NA	101,69	101	0.63
0.1	0.1	0	40 bit	16/2	96,82	113	0.68
0.1	0.5	0	40 bit	16/3	96,72	119	0.56
0.1	1.0	0	40 bit	16/4	96,62	125	0.46
0.1	0.1	0.1	40 bit	16/2	96,76	113	0.63
0.1	0.1	0.5	40 bit	16/2	96,60	113	0.49
0.1	0.1	1.0	40 bit	16/2	96,44	113	0.35

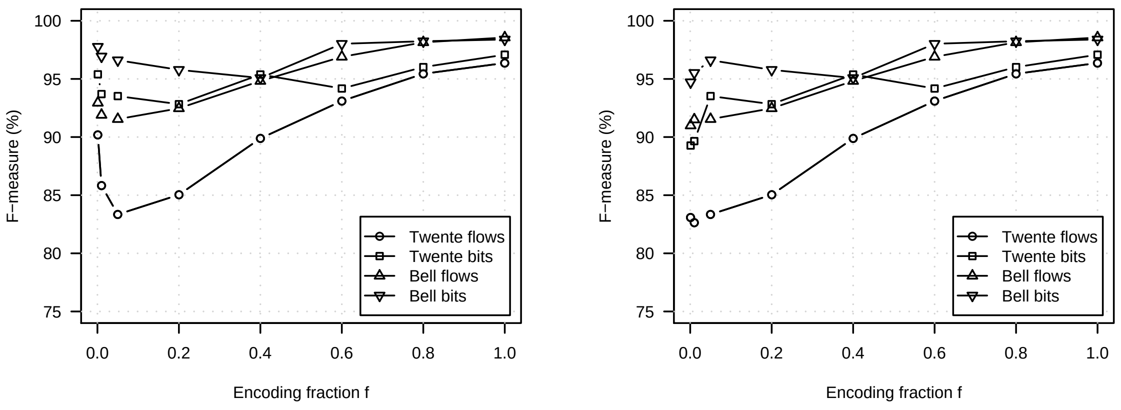
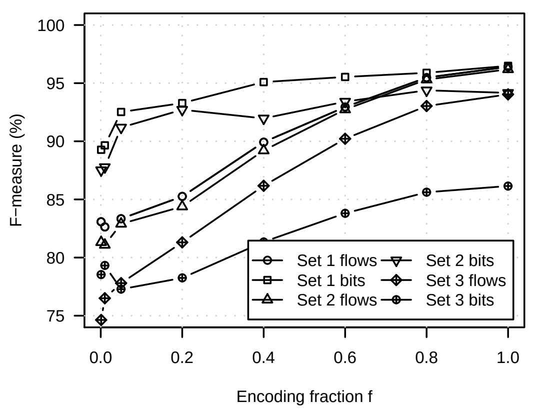


Figure B.19: Detection accuracy using DED modulation for both traces using sparse encoding (left) and using modified sparse encoding (right)



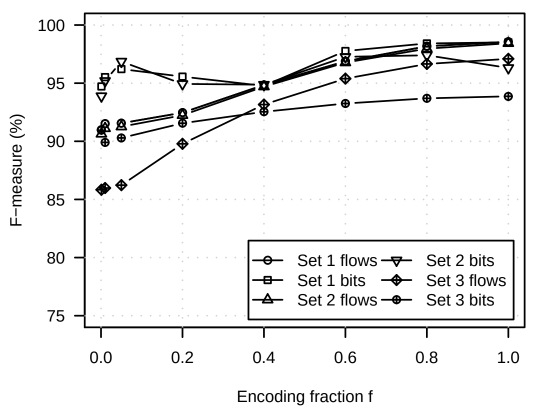
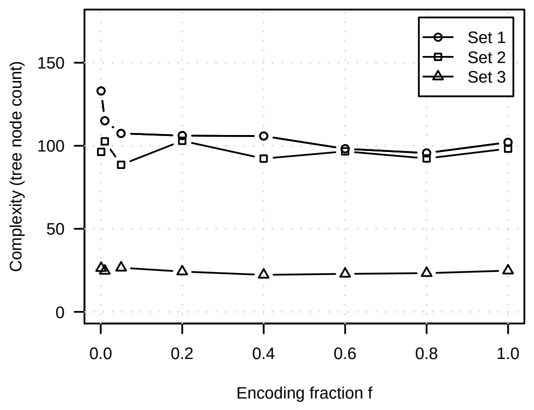


Figure B.20: Detection accuracy for the DED modulation scheme depending on the feature set for Twente (left) and Bell (right)



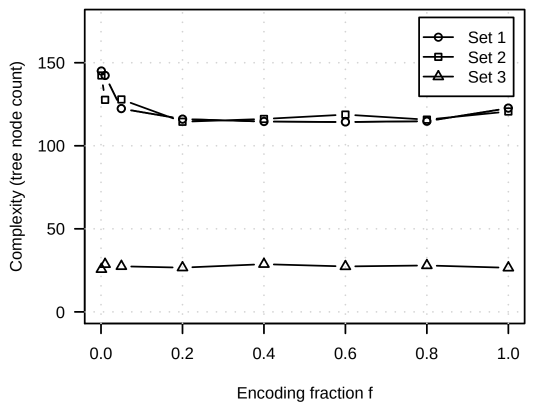


Figure B.21: Complexity of the classifier for the DED modulation scheme depending on the feature sets for Twente (left) and Bell (right)

Appendix C
Packet-timing Covert Channels

This appendix contains additional material for inter-packet timing covert channels analysed in Chapter 4 and Chapter 7.

C.1 Inter-packet time analysis

Here we show the histograms of the inter-packet gaps (IPGs) for the example flows where we show the ACFs of the IPGs in Section 4.1.

Figure C.1 shows the histograms of the IPGs of Q3 client-to-server traffic for two example clients – a local client and a remote client. Figure C.2 shows the histograms of the IPGs of Q3 server-to-client traffic for the flows for which the ACFs are shown in Figure 4.2. Figure C.3 shows the histograms of IPGs for Skype traffic for a flow measured at the source and another flow measured 11 hops away from the source. The corresponding ACFs are shown in Figure 4.4.

C.2 Sender timing accuracy

At the sender noise is mainly introduced because of inaccuracies when sending the packets. Our prototype, based on CCHEF described in Appendix A, enables a covert channel module to specify whether an intercepted packet should be re-injected as early as possible or at a later time. Packets to be re-injected later are stored in an event list. To conserve CPU time CCHEF sleeps when no events are due and no overt packets are intercepted. To maximise timing accuracy CCHEF wakes up slightly earlier for re-injection events and then performs busy waiting until the scheduled time.

This approach provides accurate timing if there are no other processes using significant amounts of CPU time. Otherwise, the timing is non-deterministic, since it can happen that CCHEF does not awake at the right time because another process is still running. One option to avoid this problem is to run CCHEF on dedicated machines that are bridged in the network close to the overt senders/receivers. Another option is to use a real-time operating system and give CCHEF a higher priority than other userspace processes.

In our testbed we used the latter approach because it requires fewer computers. We used Linux 2.6.20 with the PREEMPT patch, commonly referred to as LinuxRT [198].

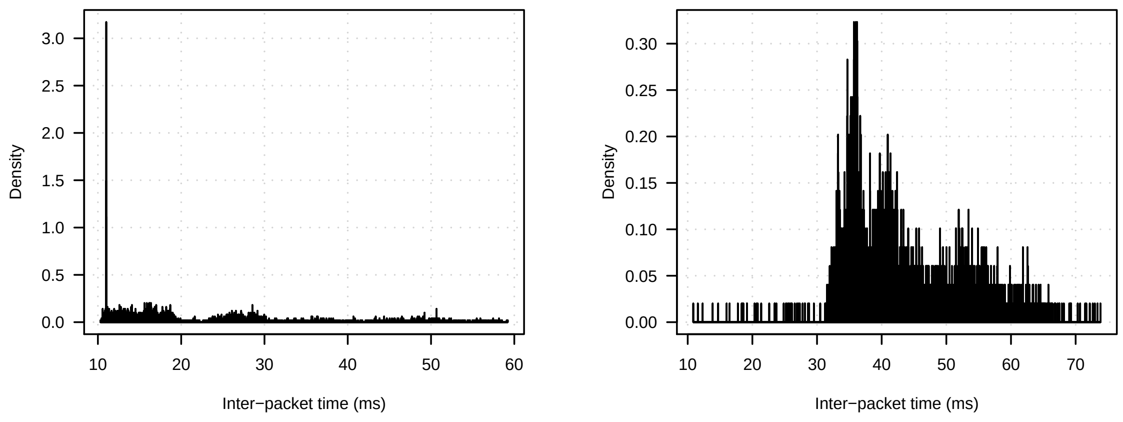


Figure C.1: Histogram of IPGs of Q3 client-to-server traffic of local client (left) and remote client (right)

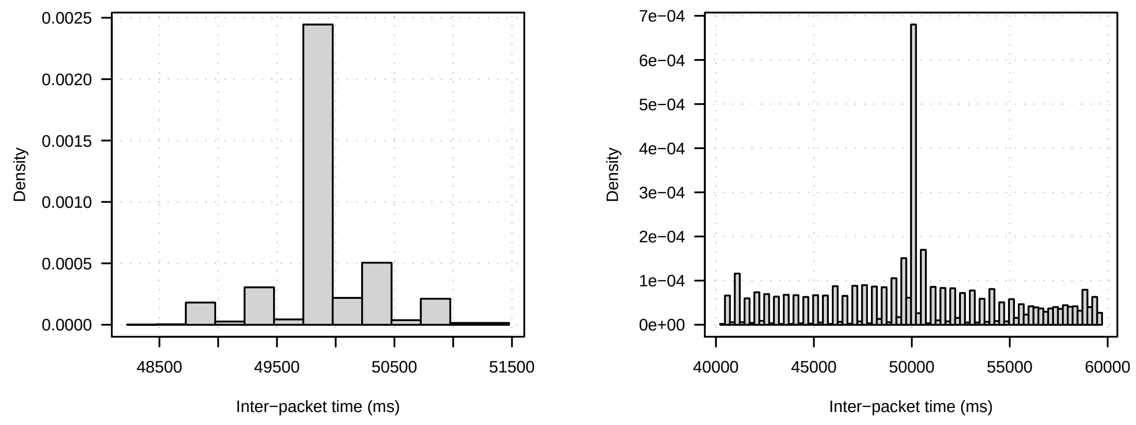


Figure C.2: Histogram of IPGs of Q3 server-to-client traffic for a server with little auto-correlation (left) and a server with moderately auto-correlated IPGs (right)

CCHEF ran as real-time process with high priority, all memory was locked to prevent page faults, and the stack was prefaulted [198].

CCHEF intercepts and re-injects packets using the Netfilter queue framework [248]. CCHEF uses the POSTROUTING hook since this hook is the last hook before a packet is sent and most of the packet processing has been done already [251]. After re-injecting a packet CCHEF tries to give up the rest of the current time slice. This allows the higher priority kernel to immediately process the packet and send it into the network. Furthermore, we set the kernel's tick frequency to 10 kHz to minimise the size of time slices. Etsion *et al.* showed that even with a tick frequency of 10 kHz there is only a small increase in context switching overhead for modern CPUs [199].

Without accurate measurement equipment (e.g. DAG cards) the accuracy of the covert sender cannot be exactly measured. However, we estimate the accuracy as follows. We

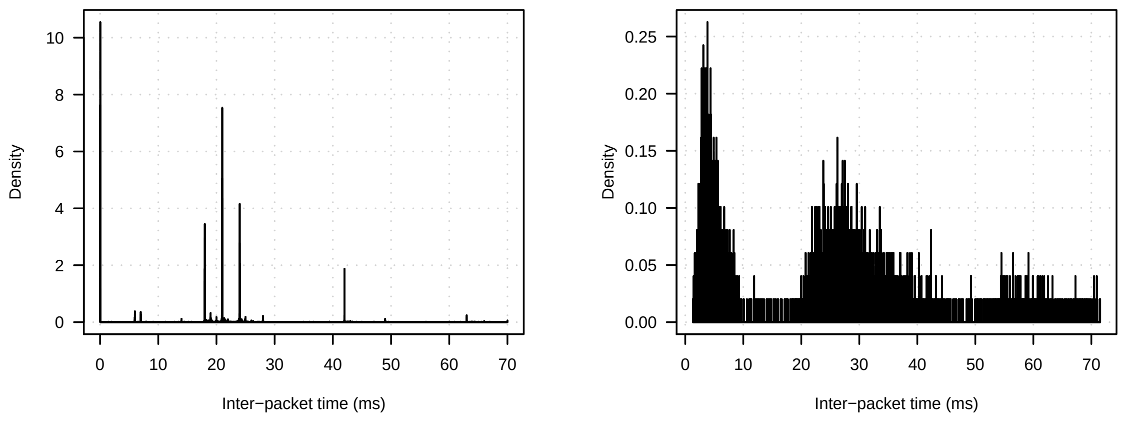


Figure C.3: Histogram of IPGs of Skype traffic measured at the source (left) and 11 hops away from another source (right)

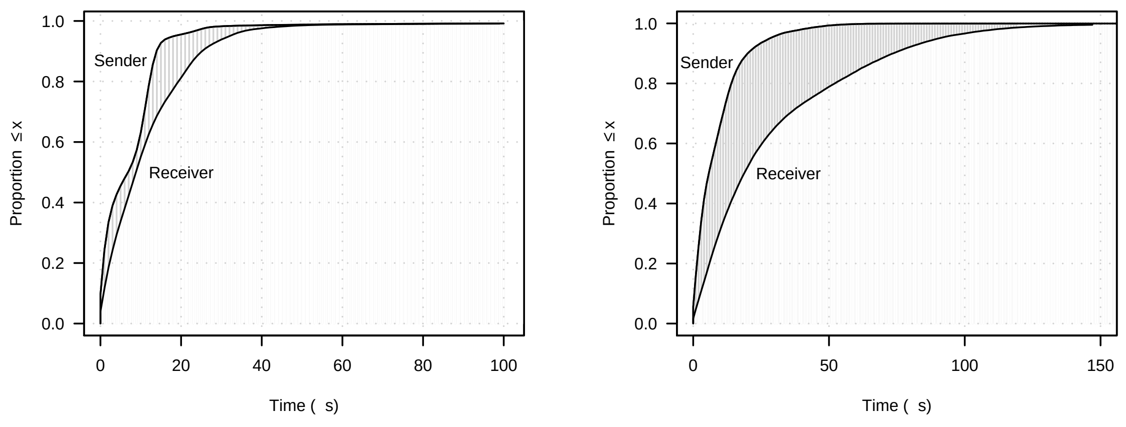


Figure C.4: Difference between send times of the covert sender and actual send and receive times measured with tcpdump for Q3 client-to-server traffic (left) and scp traffic (right)

injected the covert channel into the timing of Q3 client-to-server packets and scp packets (rate-limited to 2 Mb/s). CCHEF logged the covert sender's IPGs, and we ran tcpdump on both the sender and receiver. Both hosts were connected via a cross-over cable.

Figure C.4 shows the CDFs of the absolute differences between the IPGs of CCHEF and the IPGs measured with tcpdump at the sender and the receiver. In each graph the left curve shows the time differences at the sender, which underestimate the noise because tcpdump sees the packets before they are actually sent into the network. The right curve shows the time differences at the receiver which overestimate the noise because they also contain the noise introduced by the receiving process. The noise is bounded by the two lines (shaded area).

The results show that for UDP packets at a rate of approximately 86 pps the error is less than 40 µs, despite the covert sender and Q3 client running at 100% CPU load. For

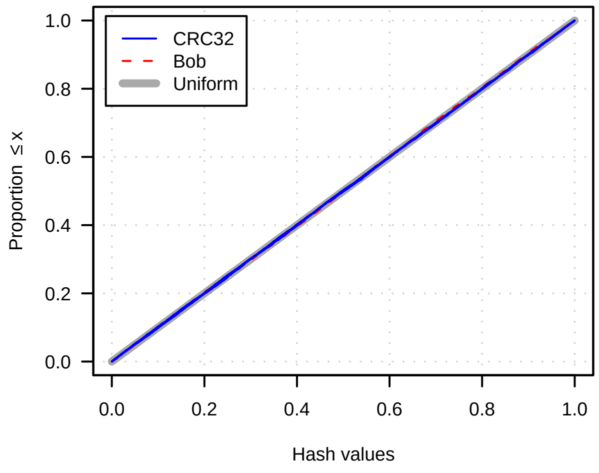


Figure C.5: Distribution of packet hash values for different types of overt traffic vs. uniform distribution

scp traffic the error is higher because of the higher packet rate, approximately 164 pps, and the larger amount of processing needed for TCP packets, but still mainly within 100 µs.

It is possible to further improve the accuracy by running the covert sender and receiver in kernel space, for example as Linux kernel modules. In previous work we showed that a kernel-based UDP sender has timing inaccuracies of less than 5 µs even at high packet rates [252]. Such a solution has potential drawbacks, such as reduced flexibility and extensibility.

C.3 Hash function output

Figure C.5 shows the CDFs of the values of the two hash functions for the different types of overt traffic, as well as the theoretical uniform distribution (see Section 4.3.3). The figure shows that for all applications both hash functions deliver almost perfect uniform distributions, since all lines are basically on top of each other.

C.4 Error rate and capacity

Here we show additional results that are not shown in Section 4.5. Figure C.6 shows the error rate for sparse encoding with model-0.75 depending on the standard deviation of the emulated delay and the maximum transmission rates for each overt traffic type. Figure C.7 shows the same graphs for sub-band encoding with model-0.75.

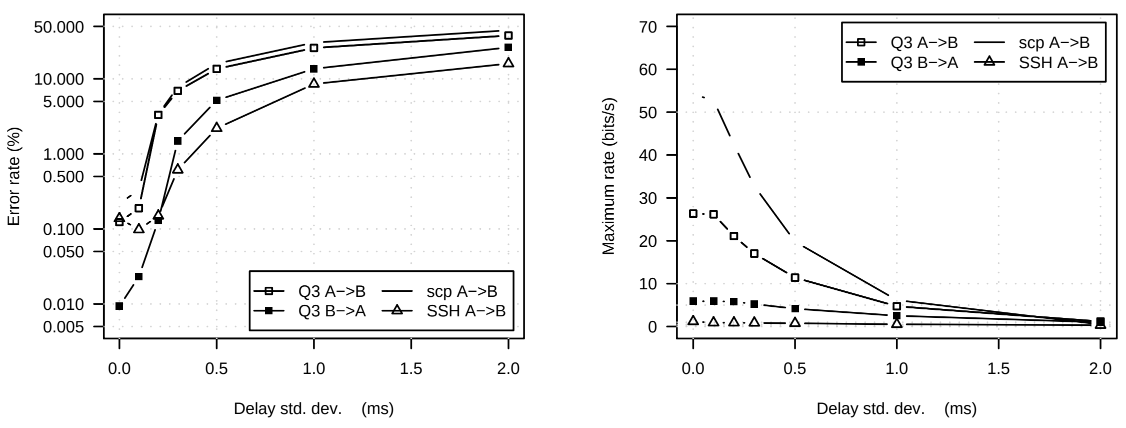


Figure C.6: Error rate (left) and maximum transmission rate (right) for sparse encoding with model-0.75 (left graph has log *y*-axis)

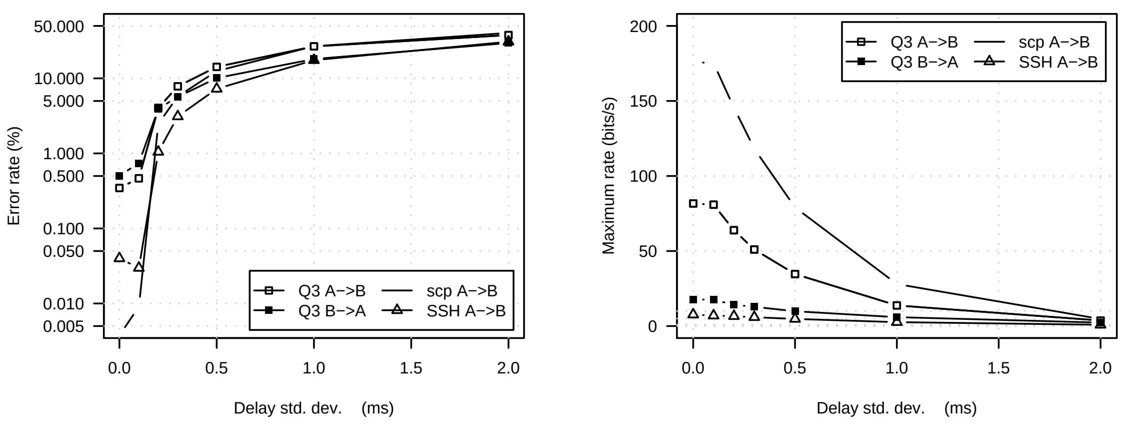


Figure C.7: Error rate (left) and maximum transmission rate (right) for sub-band encoding with model-0.75 (left graph has log *y*-axis)

C.5 Covert bit rate over time

Here we analyse the average bit rate of the channel over time. We analyse the raw bit rate after demodulation, but prior to framing and error correction. We use the data from the experiments described in Section 4.5. Figure C.8 plots the bit rate over time for Q3 client-to-server traffic and scp traffic, averaged over non-overlapping windows of 500 bits, for sub-band encoding with 0% and 1% packet loss and sparse encoding with 0% packet loss (Pareto-distributed network delay with $\sigma = 0.2$ ms and model-0.5).

For Q3 client-to-server traffic the bit rate is relatively constant for both encoding schemes even for 1% packet loss, apart from the initial connection phase and map changes

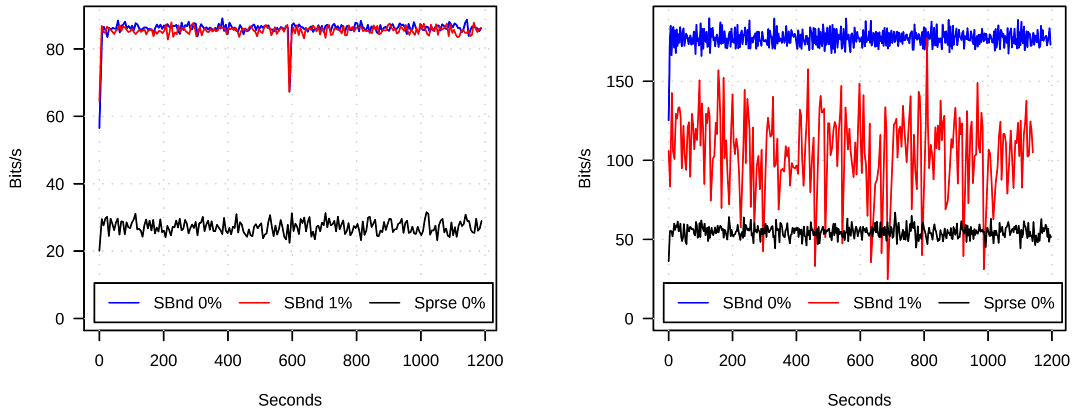


Figure C.8: Average bits per second over time for Q3 client-to-server traffic (left) and scp traffic (right)

occurring every 600 seconds[1]. Packet reordering has no effect on the burstiness. For Q3 server-to-client and interactive SSH traffic the results are similar and hence not shown.

For scp the bit rate is relatively constant for both encoding schemes as long as there is no packet loss, but there is more variation than for Q3 and SSH traffic. With packet loss the overt packet rate and hence the covert bit rate fluctuates greatly for both encoding schemes. The higher the loss rate is, the greater the burstiness is. We leave a more comprehensive study as future work.

C.6 Code parameters

Table C.1 shows the code parameters used in the different experiments. We used one set of codes for Q3, which had the highest error rates, and another set of codes for scp and SSH, since their error rates were similar. In the latter case the codes were tuned for scp, which experienced slightly higher bit error rates.

In all experiments with packet loss and reordering the outer marker code had a 40-bit preamble. For the inner marker code we state the number of bits between markers followed by the number of marker bits. For the RS code the table states *N*,*K*. Note that *K* includes the CRC-32 checksum and 8-bit sequence number part of each block. The table also shows the code rate, which was verified from the experimental data.

[1]Map changes are masked for sparse encoding due to the much lower bit rate.

Table C.1: Code parameters

Delay std. dev. (ms)	**Loss (%)**	**Reordering (%)**	**Application**	**Inner Marker**	**RS**	**Block size (bytes)**	**Code rate**
0	0	0	Q3	NA	101,91	101	0.85
0	0	0	scp, SSH	NA	101,95	101	0.89
0.1	0	0	Q3	NA	101,85	101	0.79
0.1	0	0	scp, SSH	NA	101,93	101	0.87
0.2	0	0	Q3	NA	101,61	101	0.55
0.2	0	0	scp, SSH	NA	101,75	101	0.69
0.3	0	0	Q3	NA	101,29	101	0.24
0.3	0	0	scp, SSH	NA	101,41	101	0.37
0.2	0.1	0	Q3	16/2	96,56	113	0.45
0.2	0.1	0	scp, SSH	16/2	96,68	113	0.60
0.2	0.5	0	Q3	16/2	96,42	113	0.33
0.2	0.5	0	scp,SSH	16/2	96,52	113	0.42
0.2	1.0	0	Q3	16/3	96,28	119	0.19
0.2	1.0	0	scp,SSH	16/3	96,46	119	0.35
0.2	0.1	0.1	Q3	16/2	96,50	113	0.40
0.2	0.1	0.1	scp, SSH	16/2	96,66	113	0.54
0.2	0.1	0.5	Q3	16/2	96,44	113	0.35
0.2	0.1	0.5	scp, SSH	16/2	96,62	113	0.50
0.2	0.1	1.0	Q3	16/2	96,32	113	0.24
0.2	0.1	1.0	scp, SSH	16/2	96,48	113	0.38

Appendix D
Covert Channels in Multiplayer Games

This appendix contains additional results for the covert channel in multiplayer games analysed in Chapter 5 and Chapter 7.

D.1 Maximum number of encodable bits

Here we show how the right hand term of equation 5.9 is derived from equation 5.8:

$$\frac{\max(|\delta_i(b,N_i)|)}{\max(|\delta_i(b,N_i)|)+|\Delta_i|} \le L\,. \tag{D.1}$$

The maximum absolute value for δ_i given N_i is $2^{N_i}-1$ and so it follows:

$$\frac{2^{N_i}-1}{2^{N_i}-1+|\Delta_i|} \le L$$

$$2^{N_i}-1 \le L2^{N_i}-L+L|\Delta_i|$$

$$2^{N_i}-L2^{N_i} \le 1-L+L|\Delta_i|$$

$$N_i \le \left\lfloor \log_2\left(\frac{1+L(|\Delta_i|-1)}{1-L}\right)\right\rfloor\,.$$

D.2 Making Q3 visibility symmetric

The PVS in the map file can be modified to make it symmetric using algorithm D.1. The processing time is negligible for small maps. For example, making the PVS of the standard map *q3dm1* symmetric takes less than one second on an Intel Core 2 Duo 2.4 GHz machine. While it would take longer for bigger maps the time would still be very small compared to the total time needed to build a map [253].

The Q3 server implementation also needs to be modified to make visibility in snapshots symmetric. One solution is to compute the visibility based on all clusters a player's bounding box touches (see algorithm D.2). If C_A and C_B are the set of clusters a bounding box around Alice's and Bob's player character touches, the server has to check if any

Algorithm D.1 Making the PVS matrix in Q3 map files symmetric

```
for i in 0...num_of_clusters do
   for j in 0 to num_of_clusters do
      if bit(i,j) > 0 then
         bit(j,i) = 1
```

Algorithm D.2 Making visibility symmetric in Q3 server (simple solution)

```
// build snapshot for player p
for e in 0...num_of_entities do
   c1 = get_clusters(p)
   c2 = get_clusters(e)
   for i in 0...c1.size()-1 do
      for j in 0...c2.size()-1 do
         if PVS[c1[i],c2[j]] > 0 then
            add_entity_to_snapshot(e)
            break
```

cluster in C_A is visible from any cluster in C_B and vice versa. While this solution is simple to implement, it increases the complexity from $O(n)$ to $O\left(n^2\right)$.

An improved solution keeps track of the visibility between all players for each snapshot by implementing a $p \times p$ player visibility matrix (PVM), where p is the maximum number of players. Each element (n,m) of the matrix is either set to NA (visibility unknown), to zero (player n cannot see player m) or one (player n can see player m). Before snapshots are built the matrix is initialised with NA values. Then the server builds the snapshot for each player using algorithm D.3.

The algorithm builds on four functions: `is_player(e)` returns true if `e` is a player, `get_player_num(e)` returns the number of a player entity, `is_visible(p,e)` returns true if player `p` can see entity `e` (existing Q3 function), and `add_entity_to_snapshot(e)` adds the entity `e` to the snapshot for the current player (existing Q3 function).

This improved solution should be faster than the current Q3 server implementation. The first test (Alice visible to Bob?) is still $O(n)$, but the second test (Bob visible to Alice?) reduces from $O(n)$ to $O(1)$. Since the maximum number of players is limited to 32, the amount of extra memory needed for the PVM is negligible.

We implemented the simple solution and did not measure any increase in CPU load on the server for up to three players on a small map (q3dm1).

Algorithm D.3 Making visibility symmetric in Q3 server

```
// build snapshot for player p
for e in 0...num_of_entities do
   if is_player(e) then
      x = PVM[p, get_player_num(e)]
      if x = NA then
         if is_visible(p, e) then
            add_entity_to_snapshot(e)
            PVM[get_player_num(e), p] = 1
         else
            PVM[get_player_num(e), p] = 0
      else if x = 1 then
         add_entity_to_snapshot(e)
   else
      // standard procedure
```

Appendix E
Temperature-based Covert Channels

This appendix contains additional results for the temperature-based covert channel analysed in Chapter 6 and Chapter 7.

E.1 Sending jitter and target jitter

We carried out a number of experiments in a LAN to estimate the jitter introduced by the target, the attacker's probe sending jitter and the jitter caused by timestamping of probe responses at the attacker (see Section 6.2).

To estimate the jitter on the target we used the setup shown in Figure E.1, using the same computers for attacker and target as described in Section 6.4.1. All traffic between attacker and target was measured via an optical splitter and timestamped by a high-precision traffic measurement DAG card [197] inside a separate PC. The DAG card allowed us to very precisely measure the time between each probe from the attacker and the corresponding response from the target.

Figure E.2 shows the variable part of the time differences between probe packets and corresponding response packets as measured by the DAG card (RTT jitter). This is the jitter caused by the target. It is not possible to further separate this into the jitter affecting the probes and jitter affecting the responses. The jitter is mainly 0–80 µs with a tail extending up to 120 µs.

Another source of error is the inability of a userspace program to exactly control the sending of packets. There is a variable delay between the time a userspace program executes the `send()` system call and the time the NIC actually sends the packet on the

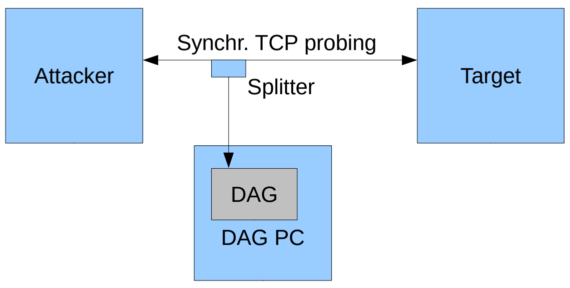


Figure E.1: Experimental setup to measure the timing jitter introduced by the target

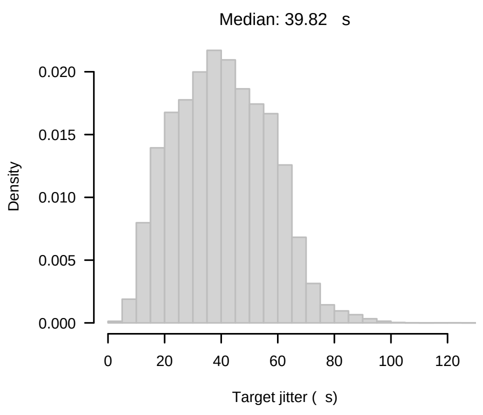


Figure E.2: RTT jitter caused by the target

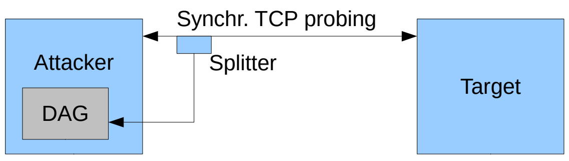


Figure E.3: Experimental setup to measure the timing jitter inherent in sending probe packets and timestamping responses at the attacker

medium. For estimating this error we used the network setup shown in Figure E.3, with the same attacker and target as before.

The attacker probed the target using synchronised sampling. An optical splitter duplicated all traffic on the network between attacker and target and fed a copy of it into a DAG card inside the attacker, whose own clock chip was synchronised with the host clock. According to the DAG statistics 99% of the time synchronisation was within ±2 µs for the duration of the experiment.

A histogram of the variable part of the time differences between the execution of the `send()` call and reception of the probe by the DAG card is shown in Figure E.4. The send jitter is mostly in the range of 0–35 µs, but there are very few large values of up to 600 µs.

This experimental setup also allowed us to estimate the jitter between the arrival of responses and when they are actually timestamped. Figure E.5 shows the variable part of the time differences between the response packet timestamps generated by the Linux kernel and the DAG card. The receive jitter is 0–60 µs and looks almost uniformly distributed.

E.2 Network jitter

Figure E.6 shows the RTT/2 jitter measured across the 22-hop Internet path (see Section 6.4.2). Despite the high average RTT of 325 ms, the jitter is relatively small and skewed towards zero. Figure E.7 shows the RTT/2 jitter measured over the PlanetLab Tor testbed (see Section 6.4.3). The jitter is considerably higher than in all other experiments.

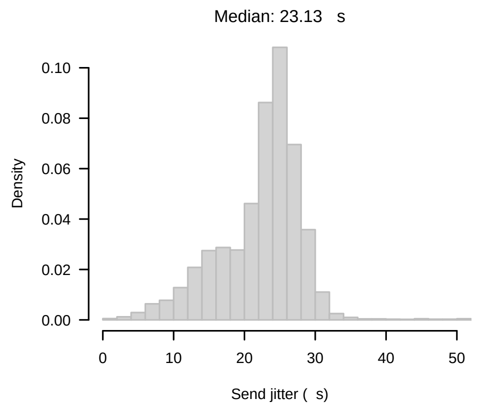


Figure E.4: Probe sending jitter at attacker (x-axis is cut off at 50 µs)

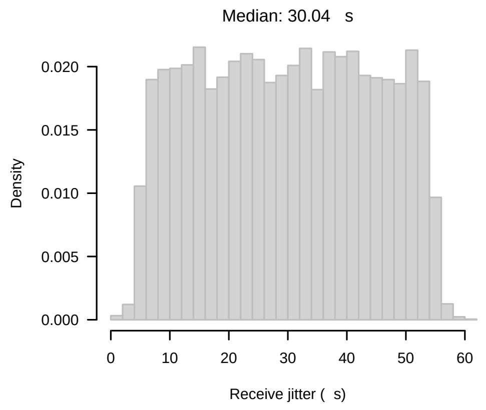


Figure E.5: Probe response timestamping jitter at attacker

E.3 Noise normality tests

We use the Shapiro-Wilk statistical test of normality to test whether the noise has a Gaussian distribution. This test performed very well when compared against other tests of normality [254]. Table E.1 shows the statistics for all data and for 96% of the data (2% outliers removed at each edge). For Intermediate 1 with outliers removed we cannot reject the hypothesis that the data is Normally distributed at 99% significance level. For Intermediate 2 we cannot draw the same conclusion, as the resulting p-values are too low.

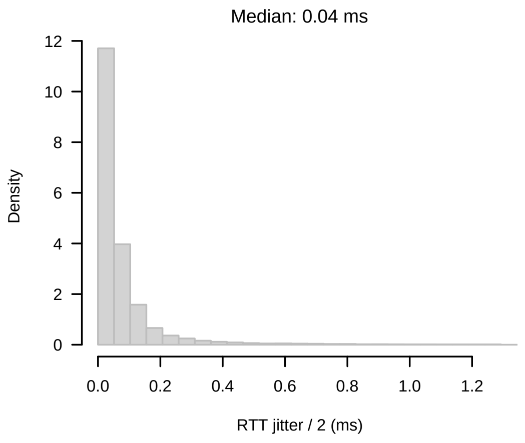


Figure E.6: RTT/2 jitter on path across the Internet

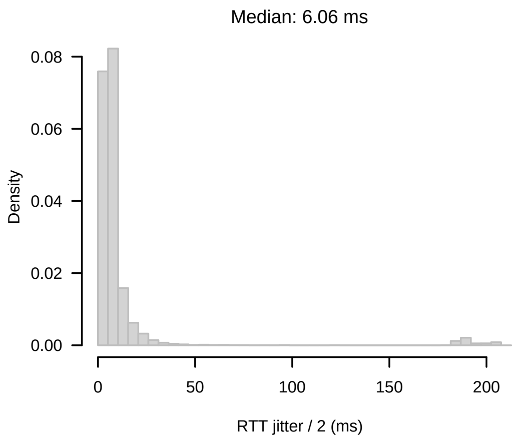


Figure E.7: RTT/2 jitter over PlanetLab Tor testbed

Table E.1: Shapiro-Wilk test statistics and p-values

	Intermed. host	**Test statistic (W)**	**p-value**
Day (100%)	1	0.973	$\ll 1\%$
Day (96%)	1	0.995	2.6%
Night (100%)	1	0.987	$\ll 1\%$
Night (96%)	1	0.996	1.5%
Day (100%)	2	0.992	0.05%
Day (96%)	2	0.982	$\ll 1\%$
Night (100%)	2	0.981	$\ll 1\%$
Night (96%)	2	0.991	$\ll 1\%$

Appendix F Netem Accuracy

We conducted a number of experiments to verify the accuracy of Netem. Our testbed consisted of two computers[1] connected by a Fast Ethernet switch. Both Linux kernel's tick timer frequencies were configured to 10 kHz. We then used Netem to emulate constant RTTs of approximately 25 ms, 75 ms and 125 ms, by setting one-way delays of 12 ms, 37 ms and 62 ms (initial experiments indicated that the actual delay is always a few hundred microseconds higher than the configured delay). We also used Netem to emulate packet loss rates of 0.1%, 0.5% and 1.0%.

We used Q3 to generate traffic in both directions. We also ran CCHEF on both hosts with a special module that does not create a covert channel, but instead for each packet logs its arrival timestamp and a packet hash, where the hash is computed as described in Section 4.2.2 and Section 4.3.3. After the experiments we computed the RTTs from the two series of packet hashes and timestamps using SPP [241]. We also computed the loss rate based on the packet hashes.

Figure F.1 shows the RTT distributions measured in the experiments. For each RTT emulated there is a very narrow distribution around a median slightly lower than the desired RTT. The median values of the three distributions are 24.9 ms, 74.9 ms and 124.9 ms. The standard deviation of each distribution is approximately 0.09 ms. The results demonstrate that the delay emulation is very accurate, if one takes Netem's 'overhead' into account when configuring RTTs.

We performed the same experiments with the kernel's tick timer frequency set to 1 kHz. The resulting RTT distributions are wider and the standard deviations increase to almost 1 ms. However, the reduced accuracy is still sufficient for the experiments with the TTL channel and FPSCC. For packet loss the results show that the actual loss rates are very much identical to the loss rates configured for Netem (see Table F.1).

When emulating delay variation with Netem we verified that packets are not reordered if a *pfifo* queue is used. Netem does reorder packets if variable delay is configured and the default *tfifo* queue is used. The percentage of reordered packets depends on the inter-packet times of the network traffic and Netem's configuration. This is the 'natural' way of emulating packet reordering, but it closely ties reordering to jitter. Reordering can only

[1]An Intel Celeron 2.4 GHz with 256MB RAM and an Intel Celeron 3.0 GHz with 1GB RAM, both running Linux 2.6.20.

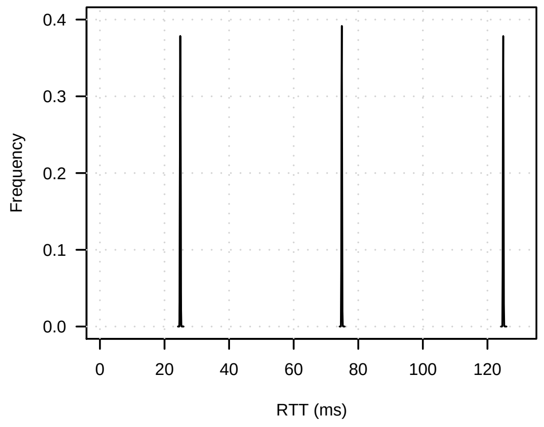


Figure F.1: Histograms of different RTTs emulated by Netem

Configured (%)	Measured (%)
0.1	0.099
0.5	0.498
1.0	0.999

Table F.1: Measured packet loss rates compared to configured packet loss rates

be increased/decreased by increasing/decreasing the overall jitter. But in our analysis of inter-packet gap timing channels we wanted to study both effects separately.

Hence we used Netem's capability of configuring the probability of reordered packets. In this case Netem delays all packets that are not reordered according to the fixed delay specified and reordered packets are not delayed. This means the actual reorder rate not only depends on the configured reorder rate, but also on the inter-packet times of the traffic and the fixed delay configured. The key difference to the first approach is that now jitter is only introduced for the reordered packets, but not for the rest. In our experiments we tuned the fixed delay for each application so that the resulting reordering rates were relatively similar.