Defending the Right to Communicate: Anonymity and Filter Traversal

&

Fair Validation of Database Replies

Der Technischen Fakultät der
Universität Erlangen–Nürnberg
zur Erlangung des Grades

Doktor – Ingenieur

vorgelegt von

Dipl. math. Matthias Bauer

Erlangen – 2004

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen–Nürnberg

Tag der Einreichung:	16. September 2004
Tag der Promotion:	3. Juni 2005
Dekan:	Prof. Dr. rer. nat. A. Winnacker
Berichterstatter:	Prof. Dr-Ing. R. German
	Dr-Ing. M. Kaiserswerth

Das Recht auf Kommunikation verteidigen: Anonymität und Filter–Unterwanderung

&

Gerechte Überprüfung von Datenbank–Antworten

Der Technischen Fakultät der
Universität Erlangen–Nürnberg
zur Erlangung des Grades

Doktor – Ingenieur

vorgelegt von

Dipl. math. Matthias Bauer

Erlangen – 2004

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen–Nürnberg

Tag der Einreichung:	16. September 2004
Tag der Promotion:	3. Juni 2005
Dekan:	Prof. Dr. rer. nat. A. Winnacker
Berichterstatter:	Prof. Dr-Ing. R. German
	Dr-Ing. M. Kaiserswerth

Inhaltsverzeichnis

Die Arbeit besteht aus zwei Teilen. Die Hauptaussage des ersten Teils besteht darin, dass geheime Zwei–Wege–Kommunikation im Internet nicht verhindert werden kann, unabhängig von etwaigen Filtermethoden. Wir zeigen wie Partner Kanäle für spätere Kommunikation auch unter aktiver Filterung anlegen können. Wir untersuchen verdeckte Kanäle auf allen Schichten des TCP/IP Schichtmodells und der wichtigsten Anwendungsprotokolle. Wir beschreiben Anonymisierungstechniken und ihre Beschränkungen in bestimmten Szenarien. Durch Kombination von verdeckten Kanälen in einem verbreiteten Protokoll und Anonymisierung entwickeln wir ein Nachrichtenprotokoll das unbeobachtbare Kommunikation erlaubt.

Im zweiten Teil der Dissertation stellen wir ein Protokoll zur interaktiven Überprüfung von Datenbank–Antworten vor. Das Protokoll ist effizient in der Anzahl und Größe der ausgetauschten Nachrichten. Eine besondere Eigenschaft ist, dass sogar die Nicht–Existenz von Einträgen effizient verifiziert werden kann.

This dissertation consists of two parts. The main thesis of the first part is that secret, two–party communication cannot be prevented on the Internet, regardless of filtering methods. We show how partners can arrange a channel for further communication despite active filtering. We examine covert channels on all layers of the TCP/IP stack and its main application protocols. We describe anonymizing techniques and their limits in certain scenarios. By combining covert channels in a popular protocol and anonymization we develop a messaging protocol which allows unobservable communication.

In the second part of the dissertation we present a protocol for the interactive validation of database replies. The protocol is efficient in the number and size of the messages exchanged. A special property is that even the non–existence of records can be verified efficiently.

Contents

Part I

Free Communication in Presence of strong Adversaries

Chapter 1

Introduction

The right to communicate is one of the fundamental human rights in the Universal Declaration of Human Rights [1], the proposed European Constitution [2], and many national constitutions, e.g., the German Grundgesetz [3]. To quote the Universal Declaration, Article 19:

> Everyone has the right to freedom of opinion and expression; this right includes freedom to hold opinions without interference and to seek, receive and impart information and ideas through any media and regardless of frontiers.

1.1 Free Communication on the Internet: a Utopia?

Ideally, communication should not only be possible, but there should be no negative consequences or fear thereof. To ensure this, the following requirements would have to be met:

1. Two parties can initiate communication without being observed or obstructed.
2. The content of their conversation remains oblique to anyone but themselves (as far as the network is concerned).
3. The very act of communication should not be observable at all.
4. Failing this, it should be extremely hard to identify the communicating parties.

The question is, whether this high ideal can be reached in the Internet.

The Internet was considered to inherently promote rather than to hinder this and related rights (the right to free expression, for example). This hope was based on the End–to–End design of the Internet Protocol, which transports arbitrary data between each pair of end–points regardless of application–layer content. In chapter 2 we will discuss some of the protocols proposed then, which were to enable free expression of thought and anonymous publication.

The hope turned out to be misplaced, however. On the contrary, digital, text–based communication is much easier to monitor than speech over

telephone or letters. If surveillance can be automated and becomes cheap, then outright Orwellian laws can be implemented without much resistance. To name a recent example, a European law mandating traffic data retention would have been unthinkable for monitoring the exchange of handwritten letters. The monitoring of communication patterns is very much facilitated by the Internets standard protocols. Such patterns allow estimates about the structure of groups of people who share interests, decidedly a invasion of privacy and the right to free association.

The threats to free communication can be classified by their source: political pressure, economic interests and technical misdesign. We will discuss the first only shortly and expand on the latter two.

Political Pressure

Aided by the fact that national telecoms make up much of their countries Internet backbone, regimes of countries like China or Saudi–Arabia have access to the data streams of almost all their citizens. This allows them to filter the exchange of ideas, block access to information, and to punish people for stating their opinion in private [4].

Economic Interests

In countries less notorious for suppression, free enterprise is trying to amass as much data as possible about their customers and their habits. The goal behind most of these attempts is to price discriminate [5], i.e., to charge different people different prices for the same good, according to the customers willingness and ability to pay. Economists have discussed price discrimination from various angles and for diverse industries, see for example [6]. Proponents argue that price discrimination increases efficiency of markets, i.e., more people willing to buy a certain product will actually be able to do so. Opponents point out that

1. plainly "unjust" prices are unpopular in most societies.

2. discriminatory pricing is hard if people communicate about the prices.

3. the right of "second sale" has to be abolished for goods sold under discriminatory pricing.

4. the proponents' arguments imply that money is always invested best by the supply side and not by the customers.

Commercial consortia and individual corporations have in recent times used pressure to suppress criticism of their products or conduct on the Internet. Examples are the Church of Scientology's attacks on its critics[7] or the Secure Digital Music Initiative's proceedings against the scientists that found flaws in their copy–prevention scheme [8].

Technical Misdesign

The perceived scarcity of IP addresses motivated technical decisions which make free, unmitigated communication harder. Until the early 90's most machines connected to the Internet provided services for everyone, namely for remote logins, administration and file transfers. It was standard to be reachable from every machine on the net. This End–to–end paradigm implicitly enabled free communication, by allowing everybody connected to communicate over arbitrary protocols with everybody willing to listen.

The explosive growth of the Internet in the mid–nineties caught the designers of the net quite unprepared. The early distribution of addresses in extremely large blocks (class–based addressing [9, pp. 140–141],[10]) mapped the theoretically possible 2^{32} addresses very unevenly to a few institutions, most of them in the U.S. The following split–up of these large blocks into smaller subnets (Classless Inter Domain Routing [10]) dispersed thousands of small address–blocks over many countries. But the numerical relations between those address–blocks did not mirror the topological relations, which made huge routing tables at the backbones unavoidable.

Dial-up and Dynamic Addresses

When millions of users started to connect their machines to the net, a scarcity of globally routable addresses became apparent. This was solved in an ad–hoc manner by providing dial–up users with *dynamic addresses*. Dynamic addresses are assigned to the user's machine during the link–layer setup [11, 12]. The idea is that there are never more than a certain number of users online, and the address space is divided among them. With most dial–up providers, this means that after every reconnect of the link–layer connection, the machine will have a different address. This is not widely considered a problem, since the users are supposed to run only client software on their machines, which will not be contacted by anyone. This assumption silently discards the right to communicate without a mediator in between (point 1 in the list above).

Advent of DSL

With the advent of digital telephony (as in ISDN or DSL) a machine can be online continuously without tying up much of the telecommunication provider's infrastructure. This led to an even greater demand for addresses, since the concept of connection hang–ups, as in the above model, is not meaningful for the new protocols, where a single switch (in ISDN) can handle a large number of connections in parallel.

To overcome the scarcity of addresses, a mechanism called *Network Address Translation* (NAT) is often employed. Here, a whole subnet masquerades behind a small set of globally routable addresses, which are assigned to the uplink interfaces of a gateway machine. If a machine in the subnet sends a datagram to an outside host, the IP header of the datagram is manipulated at the gateway and the source address of the machine replaced with one of the routable addresses of the gateway. The same is done with the destination address of answering datagrams from the outside. To decide which datagram is really addressed to what internal machine, the gateway has to examine the addresses of the outside hosts and the port numbers of the connections [13]. This obviously works with the transport layer protocols UDP,

TCP and SCTP [14, 15, 16], where the datagrams carry enough state information to distinguish between different sessions belonging to different inside machines. With stateless protocols as raw IP or with opaque protocol headers as in IPsec's ESP [17] it is not feasable. The mapping between internal and external addresses and port–numbers is initiated by connections from the inside, so it is not possible to open a connection to hosts in the subnet from the outside. Again, this is not seen as a problem, since clients almost always initiate the connections to the server, and server processes are not expected to run on machines *at the edges of the Internet*. This again restricts the possibility to communicate over freely chosen protocols.

In the original design of the Internet the majority of hosts would not route traffic for other hosts. But all host were able to connect to each other. Practically any functionality could be implemented at the ends with "stupid" routers moving the data regardless of the content or higher–layer functionality (the *End–to–End* design, see [18]). This design allows the introduction of new features at will at the edges of the network, without changes to the expensive infrastructure at its core. With the introduction of "smart" middle–boxes such as NAT, firewalls and preferences for certain types of traffic, the end–to–end paradigm is abandoned. This forces communication to flow through fewer channels, and necessitates intermediaries.

The Internet Architecture Board remarks in [19]:

> It remains a fact that today, NAT inhibits progress beyond the simple Web client/server model, and that the only well understood solution that definitively fixes this problem is general adoption of a larger address space.

The designated successor to the current Internet Protocol, IPv6, was designed to make such contraptions as NAT unnecessary [20]. The address space is vastly larger (2^{128} addresses) than in IPv4, so scarcity will not be a problem (most IPv6 standards assume that the smallest subnets will have 2^{64} possible addresses). Despite IPv6's original aim of simplicity, common off–the–shelf routers do not support IPv6, yet. When and if big service providers will offer IPv6 connectivity remains to be seen, at the time of writing, not a single major Internet Service Provider (ISP) for end–users in Europe and in the United States does. This seems at least partially caused by economic reasoning: if addresses are scarce in IPv4, an ISP can charge additionally for routable and unique addresses and for stable name–to–address mappings. Thus there is no economic incentive to provide all customers alike with large numbers of globally routable, unique addresses for a fixed sum, as would be the case with IPv6. We do not definitely know whether this is really the cause of the ISPs' reluctance to offer IPv6; it remains to be seen if the now experimental IPv6 network will attract enough commercial interest to make a majority of users, developers and businesses change to the new protocol.

Lack of support

Lack of monetary incentive for ISPs

Economic interests and the mentioned technical misdesigns allow establishments of business models based on artificial scarcity. Shortly after the Internet started to grow explosively, economic models were developed to take

advantage of the millions of potential customers now online. Since the perspective on data networks from an economic view is certainly different than from a purely technical view, we will make a short excursion into economic theory (for a thorough introduction to neo–classical macroeconomics, we refer the reader to [21]).

Excursion: Economics of data networks

We will shortly examine the position of ISPs and content providers with regard to their goods and the markets in which they operate.

The goods that an ISP has to sell is basically data transport at varying bandwidths. Data transport is a *commodity*, like electricity. This means that the price can only depend on the quantity and thus the ISPs' market place is a market with *perfect competition*. Such markets show the tendency for prices to fall until the businesses can just survive [1]. This phenomenon can be observed readily in the German telephony market since de–regulation. The ISPs' goals will therefore be to differentiate their goods artificially and sell basically the same goods at different prices. This method of discriminatory pricing allows them to charge the customer by the individually *perceived value* of the goods, as opposed to the actual costs of producing it. We will discuss how ISPs accomplish discriminatory pricing shortly.

In the case of content providers, the goods are digitally encoded informations. Information per se is a *public good*. This means it can be *shared* without losing its value. Since digitized information can be replicated arbitrarily at practically no cost, the price per copy — in a market with perfect competition — should gravitate towards zero in short time. From classical macro–economic models this must be concluded to be detrimental for various reasons (see for example [22]). Such theories suggest *artificially restricting* the access to public goods, in the hope that the ensuing scarcity will raise prices and thus motivate people to produce such goods for gain. A striking example of this thinking is the wording of article I, section 8 of the U.S. constitution:

> The Congress shall have power to promote the Progress of Science and useful arts, by securing for limited times to Authors and Inventors the exclusive right to their respective Writings and Discoveries.

Most *e–business* models use the client–server paradigm, where a set of machines, the *servers* is dedicated to provide services for another set, the *clients*. The client–server paradigm is firmly embedded into the UNIX networking application programming interfaces and the phrasing of most Internet Engineering Task Force (IETF) standards.

The e–business perspective on networking changes the paradigm to *disjoint* sets of machines, a slight change in interpretation, so that it might be more appropriately named the Content Provider–Consumer model. The designers of

[1] In Adam Smith's "The Wealth of Nations" *all* markets show this tendency. The beneficial "invisible hand" of neo–classical macro–economics depends strongly on this.

such models use verbiage that avoids notions of "communication" or the actual address or location of "services", but speak of "delivery", "content" and "distribution" (as in leaflets, not processes). Our — somewhat polemic — view is that in this model, the goods (information) flow from the supply (content providers) to the demand (customers) and money flows back over some side–channel (money transfers, credit cards, ...). Since information is a *public good* as explained above, this model implies that the clients do not communicate, or else the information expensively supplied through one server could be redistributed for free by the clients. Generally, in the e–business model, control over the path that information takes from the servers is desired. This is expressed by the recent call for the adoption and legal support for so–called Digital Rights Management systems.

Service providers have adapted to the needs of e–businesses' servers with special offers such as co–location of machines at the provider, supplying fault–tolerant hardware, connectivity and power–supplies, ..., all of which are, of course, expensive. This seems justified since the servers are supposed to somehow generate revenue and profits. For the ISPs it is essential to separate strongly these high–quality services from the standard, low fee services provided for the majority of users. Since the standard Internet protocol suite allows connections from all hosts to all hosts and all ports to all ports, the low–quality service has to be a filtered, watered–down subset of TCP/IP. As a consequence, dial–up ISPs start to charge their flat–rate users additionally if they find them running any server processes, even standard ones such as `ssh` which can hardly be used for economic gain [23]. Other measures taken to "differentiate" between costly services for content providers and cheap ones for consumers involve firewall rules to make certain ports or whole protocol sets unavailable on dial–up machines, unless the customer pays additional fees. Examples:

Artificial Scarcity

- A local provider, Nefkom, at the time of writing blocks port 25 for all its customers, making it impossible to run SMTP servers.

- A Belgian service provider intentionally blocks all non–TCP, non-UDP, non–ICMP traffic to its customers, thereby preventing them from building IPsec Virtual Private Networks or joining the emerging IPv6 network[2].

The particular design of Asymmetric Digital Subscriber Lines (ADSL) is another case of such crippling technology, i.e. a much smaller traffic capacity for data from the clients than for data from their uplink provider. The maximum bandwidth for communication between hosts at the consumer end of ADSL is bounded by the uplink speed and thus restricted to a fraction of the technically possible (and paid for) speed of the line.

To illustrate the shift to controlled or mitigated communication, we describe the development of e–mail as an example.

[2]Personal communication with Wim Vandeputte at the Linuxtag, July 6th, 2001

1.1.1 Example: E–Mail

E–mail is one of the most useful services on the Internet and the most widespread means of communication on the Internet. Millions of people rely on it for business and private communication. E-mail addresses are globally unique (per RFC 822 [24]), but it takes effort to make an E–mail address known to customers and friends. Around 1995 almost all users with Internet connections had their messages delivered to their machines, either by SMTP or UUCP. At the time of writing millions of users do have cheap and fast Internet connection but cannot run a SMTP server due to the architectural miscarriages listed above. Instead they obtain accounts at web–mail providers like `hotmail, gmx, yahoo, ...` or mail–hosting services and download their mails (by HTTP or POP3/IMAP [25, 26]) from central servers. This is basically a store–and–forward network approach as in UUCP [27], which was wide–spread in the 80's, when connectivity was slow and extremely expensive (in UUCP e–mail the e–mail's address contained a list of machine names like `jones@umich!unido!erlang`; at a certain time, the local machine would forward the message to `erlang`, which would in turn connect later to `unido` and so on. The connections were very temporary, a mail from Erlangen to an Australian university could take up to several days, depending on the number of hops on the path and the frequency of their connections.). Should one of the e–mail providers go bankrupt or experience a major outage, millions of people would be unreachable and would have to start redistributing new addresses to all their contacts. It is comparable to randomly changing the telephone area code of a whole region. If somebody hosted a mailing–list at such a service (e.g. at `groups.yahoo.com`), an outage would disrupt communication of whole communities, the members of which do not necessarily know each others e–mail addresses, stopping them from continuing their discussions by other means. Apart from being an unnecessary point of failure, these centralized services can monitor the content of e–mails, record traffic patterns and apply censorship.

Web–Mail

1.2 Structure of this part of the Thesis

To satisfy the requirements at the beginning of the introduction, anonymity or untraceability of the underlying protocol is necessary. Chapter 2 will give an introduction to the concept of anonymity and mechanisms to provide it in networked scenarios. To make initiation unblockable, special mechanisms are presented in Chapter 4. Unobservable communication is possible over so–called covert and subliminal channels; Chapter 3 introduces the concepts and examines the layers of the TCP/IP stack for such channels. In Chapter 5 we combine the findings and present a messaging protocol based on HTTP, which allows unobservable — and therefore free — communication in presence of a strong adversary.

Chapter 2

Anonymity and related Concepts

When two partners communicate, their discussion should be free from fear of repercussions from third parties. This implies that neither the content of their communication nor the fact that they communicated should result in disadvantages for any of them. The first requirement can be ensured somewhat by encrypting the content, but the second is not so easily met. Hiding the fact with whom one communicates means to communicate anonymously, a concept quite natural in real–world encounters. But networked environments require identifiers of end–points for transportation of messages. To hide the true end–points in such an environment forces the users to employ rather complicated protocols.

In this chapter, we give an overview of such techniques in literature and existing implementations that try to provide anonymity or pseudonymity for communication channels. For clarification, terms and notions in the field of anonymous communication are defined first. The mechanisms explained here will be used and adapted in following chapters.

In the second part we examine unreliable broadcast schemes for their suitability as channels for certain anonymity providing protocols. We prove that time–out driven re-send mechanisms as used in TCP destroy the strict anonymity property of such protocols.

2.1 Terminology

The terms used in this thesis follow mostly the lines laid out by Köhntopp and Pfitzmann in [28].

Anonymity is the state of not being identifiable within a set of subjects, the anonymity set. The *anonymity set* is the set of all possible subjects who might cause an action. There are different anonymity sets for sending and receiving in most protocols. Anonymity increases with the size of the respective anonymity set.

Consider, e.g., a broadcast protocol: The set of participants in such a protocol is identifiable from analysis of traffic patterns and content, since they all get the same messages. A sender can easily be identified, because

the message originates from him; thus the sender anonymity set contains just one item, i.e. there is no sender anonymity. Receivers are not identifiable, because of the broadcast property; the receiver anonymity set is the set of all participants; if there are more than two of them, this provides receiver anonymity.

Unlinkability is the property of a protocol that an observer learns nothing about the relations between the subjects, objects and/or actions of the protocol by observing a run of the protocol. Message unlinkability, for example, is the property that an observer cannot tell if two messages were sent or received by the same party.

To quote Köhntopp and Pfitzmann [28, p. 2]:

> If we consider the sending and receiving of messages as the items of interest (IOI), *anonymity* may be defined as unlinkability of an IOI and an identifier (ID) of a subject.
> (emphasis in the original)

Unobservability is the unlinkability of items of interest. This implies that messages are undiscernible from random noise. Examples are Chaum's Dining Cryptographers protocol (see section 2.2.6 below), or sending spread spectrum encoded messages below the natural noise of several broadcast bands on the physical layer. In chapter 5 we present a new protocol that achieves unobservability.

Pseudonymity is the use of pseudonymous IDs in communication. One can stay pseudonymous only as long as one's various pseudonyms and other IDs cannot be linked.

2.1.1 Definition of Attackers

Before we can start to project protocols, we should define the capabilities of the attackers against which we will check the resistance of our protocols. If we allow an attacker to examine the memory of every connected host, for example, then the design of an anonymity providing protocol becomes nearly impossible.

We will consider the following attacker models:

Passive observer This attacker can merely tap a small number of network connections. In most legal systems, this models the (legal) capabilities of law enforcement personnel.

Adaptive local attacker This attacker can take over arbitrary routers, but only one at a time. On the routers under her control, she can drop or modify incoming datagrams. This models a somewhat glorified hacker.

Global attacker This attacker is able to see *all* network traffic in the routing structure down to the last router. This does not model any realistic attacker, but is nevertheless often the attacker model in research papers on anonymity.

2.2 Methods

We will now describe protocols from literature that try to provide unlinkability, anonymity or pseudonymity. To show that this remained not in the purely theoretical, we describe implementations of some of the protocols.

2.2.1 Encrypted Broadcast

The simplest protocol that provides receiver–anonymity is a combination of broadcast channels and public key cryptography.

Setup All participants in this protocol join a broadcast channel. They make their public keys known to each other over an authenticated channel. After that, every participant continuously listens for broadcasts.

Protocol for receiver–anonymous sending The sender encrypts the message with the public key of the intended receiver. She then broadcasts the encrypted message on the broadcast channel. All participants receive the message[1]. They then proceed to decrypt it with their respective secret keys. If the decryption function returns an error, the receiver interprets the message as not intended for him or her and discards it. At most one receiver will successfully decrypt the message.

This is the basic idea behind $\mathcal{P}^5$ [29], where additional encryption steps and dummy messages (random noise disguised as a message) are used. We will examine necessary conditions for broadcast–based protocols in section 2.4.

2.2.2 Chaumian Mixes and related protocols

In his seminal paper [30], David Chaum describes a protocol that guarantees unlinkability of sender and receiver. In addition, there is no detectable relationship between any pair of messages. The class of protocols including Chaum's original version and derived protocols is called *Chaumian Mixes*.

Simple Chaumian Mix

A graphic description of Chaum's mixes: The sender puts the message (on paper) into an envelope, seals it, and puts the address of the intended receiver on the envelope. She then puts this envelope into another envelope, seals this, and puts the address of a first intermediary on the outer envelope. She repeats this step for some time and puts the result in the postbox. The addressee of the outmost envelope opens it, sees that it contains another envelope not addressed to him, and puts this inner envelope in the postbox. This process of opening and re–submitting to transport repeats until the innermost envelope arrives at the receiver, who will get the real message [2].

[1] this assumes *reliable* broadcast.

[2] This metaphor ignores the *costs* (stamps) of transporting messages. There are in fact proposals to use a kind of electronic stamp to stop certain attacks on mix networks, see [31, 32].

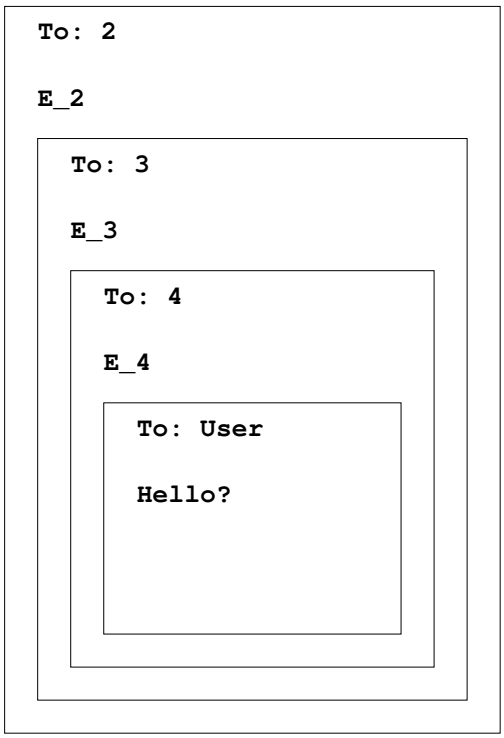

Figure 2.1: A message encrypted for node 1 . . . 4 in a Chaumian Mix

A more technical description makes clear why the intermediaries cannot open the sealed inner envelopes to get at the receiver's address or the message itself:
A group of *nodes* $n_1, \ldots, n_k$ announce their public keys $P_1, \ldots, P_k$ to each other over some authenticated channel. If one of the nodes is instructed to send a message m anonymously to node n_i, it calls r, a random number generator, l times to select a subset of nodes $n_{r(1)}, n_{r(2)}, \ldots, n_{r(l)}$ and their respective public keys $P_{r(1)}, P_{r(2)}, \ldots, P_{r(l)}$. These keys, together with the addresses of the $n_{r(1)}, \ldots, n_{r(l)}$, define a path. The sender encrypts m, the message, $l + 1$ times, in the following manner:

$$c_1 = (a_{r(l)}, E_{P_{r(l)}}(a_{r(l-1)}, E_{P_{r(l-1)}}(\ldots a_{r(1)}, E_{P_{r(1)}}(a_i, E_{P_i}(m)) \ldots).$$

where the a_j is the address of the node n_j. In other words, the message is encrypted for the final recipient, the output of that is combined with the address of the final recipient, then the next–to–last node on the path is taken as final recipient and the procedure is repeated, until the first node on the path is reached.

Each node on the path, upon receiving the encrypted message, decrypts it, thereby learning the address to which to send the rest of the decrypted message. Every node on the path only learns the addresses of the previous node

(from which it received the message) and the following node (whose address is in the decrypted message). The intermediate nodes and the receiver cannot identify the sender or determine the final receiver, because the previous node could be just another node on the path of a message, and the same is true of the following node.

2.2.3 Extensions of Chaum's Mixes

The original mix design is prone to various attacks for different kinds of adversaries [33, 34, 35], which are now described briefly.

Passive adversaries

A global observer as defined in section 2.1.1 can use the following characteristics of traffic in mix networks to determine the originator and destination of a message:

Size Messages get slightly shorter by decryption (one layer of encryption is stripped off, removing `pubkeysize + cipherblocklength + header-length` bytes). So messages entering and leaving a node can be correlated by their approximate sizes.

This can be avoided by always sending fixed–size messages and requiring additional random padding after every decryption. The first requirement however adds a lot of complexity, since messages longer than the fixed size have to be split into several messages, which must be re–assembled at some point. Note also, that sending off hundreds of fixed–sized messages along the same path in the mix network produces obvious traffic patterns.

Order If messages are sent out after decryption in exactly the same order as they arrived, tracing a message is easy, even if the enhancements above are integrated into the protocol.

To avoid this *random spooling* is employed. Incoming messages are decrypted and placed on a list. Once the number of entries in the list reaches a defined threshold, the entries are randomly shuffled and sent out in one burst. This was suggested by Lance Cottrell in 1994 [34].

Membership of sender/receiver to the set of mixes The anonymity set of Chaumian mixes contains only the nodes themselves. The current `mixmaster` re–mailers [36], however, are also used by senders/receivers outside this set. An observer who can examine all the traffic to and from all mixmaster nodes can make good guesses about the relationship of messages entering and leaving the mix network set. This is because people communicate only with a small set of other people and the different sets (cliques in terms of graph theory) have no big intersections. The only remedy for that problem is to require everybody to run a mix node, if they want to use the mix network.

Timing Let us assume that the attacker can observe the ingress of a message into the mix network. If she knows the expected latency of every link in that network, she can statistically infer the probabilities that observed outgoing messages are the processed original message. If the delay of the message is bounded (as it is in practice, by the impatience of the sender), then certain paths can be excluded from inspection by the attacker.

Kesdogan et al. suggested what they called "Stop–and–Go Mixes"[37]. The mix nodes in their scheme wait for random intervals between sending messages.

There has been a lot of research on how to send the messages from the spool, we refer the reader to [38] for an overview.

Active adversaries

An active attacker is allowed to manipulate messages, to install fake nodes and to send messages herself in addition to tap arbitrary communications. This attacker model goes beyond our definition of an adaptive attacker. Active attacks on simple Chaumian mixes include:

Timing The possibilities of a passive attacker can be enhanced, if the attacker can manipulate the latency of links, by introducing errors, for example. Most users of the network probably will then choose paths along "fast" links. This can improve the attacker's chances to successfully attack the unlinkability of messages.

Node flushing If the nodes send their spool in one burst after the spool grows to a certain size, say k messages, then the attacker can wait for a message to enter the node after a burst (so that there is only one message in the spool). She can then construct and send $k-1$ messages through the node. Since she knows to what each of her own messages decrypt, she can scan the k messages in the burst for the single one she did not construct, and try to follow the path of that message. If only a small number of messages traverse the mix network, the node flushing attack can be very efficient. This attack can at least be obstructed by sending additional dummy messages in the burst. Stop–and–Go mixes as described above also protect against this attack as long as there are still genuine messages going through the nodes.

Replay Attacks To determine the final recipient of an observed message on its way to a specific node, the attacker resends many copies of that message very fast to the node. She then watches the traffic patterns of the whole mix network. Since all the copies will take the same way through the net, peaks in communication will appear one after one between the nodes on the path of the message. The node where the flood stops is the receiver. This attack can be avoided by standard techniques. One such technique is to include unique ids in the messages and to keep a log of recently seen ids in the nodes. For each message, the nodes check if they have seen the message's id recently, and if so, they discard the message.

Subverting the Network If an attacker can install a lot of nodes and by some means make other nodes unusable, then most of the traffic will flow through nodes she controls. She can then implement timing analysis on the information she gets from the fake nodes.

Disruption As every node on a path is a single point of failure, and diagnostic messages to the sender are impossible by design, even a small portion of defective nodes can cut off a large number of paths. If the paths are chosen at random, the portion of correctly working hosts is P and the average path length is l, then the probability of a broken path is $1 - P^l$. If, for example, 20% of the hosts are defective and the path length is 3, then the probability of sending a message into oblivion is higher than 51%. If an attacker can introduce hosts into the mix network or attack existing nodes in the network by whatever means, this fragility becomes a big problem.

2.2.4 Implementations of Mixes

The *Cypherpunk* community [3] started as early as 1993 to implement Chaum's ideas. The first running network of so–called *remailers* used a protocol now called Cypherpunks Type I, which we describe below. In 1994, a new protocol emerged, mainly to address the message–size attacks mentioned in section 2.2.3. The new Type II protocol is used on a network of about 40 nodes at the time of writing. We will now briefly describe both protocols.

Type I Remailers

Type I remailers use Pretty Good Privacy [40] for encryption and special headers inside the messages for specification of handling. These headers follow an initial `::\n\n` and themselves end in a double–newline. The headers are

`Request-Remailing-To:` or `Anon-To:` contains the next hop in the chain of remailers, or the final recipient.

`Encrypted: PGP` indicates that the body is PGP encrypted.

`Subject:` is the subject header of the outgoing e–mail. Inside the remailer network normally set to `None`.

`Null:` tells the remailer to discard this message. For dummy traffic.

`Cutmark:` everything following a single line with these characters will not be forwarded. For padding.

`Latent-Time:` instructs to keep the message in the spool for this long before sending it.

[3] for a history of the movement, see [39].

When the remailer receives a message, it inspects the remailer headers. If the body is PGP encrypted, it decrypts and repeats inspection of headers in the decrypted part. The body of the message is forwarded as specified in the headers.

Note that apart from the `Anon-To` header all headers above are optional. Users who forget to specify `Latent-Time` or `Cutmark` with appropriate parameters, will have their anonymity severely reduced.

Type II Remailers

Type II remailers, also called `mixmaster` remailers, use a totally different format. Here only the headers are encrypted with public keys. For each remailer on the path the header contains a secret key with which the body and the headers for the next remailer are decrypted. The sender splits the message into equally large blocks (20 Kilobytes) which are sent separately. The last remailer on the path uses a message ID header to re-assemble the blocks for final delivery. At each node on the path, the messages are padded to the fixed size with randomness to thwart attacks exploiting message size characteristics. In order to stop replay attacks (see section 2.2.3), every remailer stores recently seen IDs and discards messages that carry any such ID.

At the time of writing, a third version the of remailer protocol is being developed, see [41]. It will support mechanisms for replying to anonymous e–mails, as well as enhancements to secure the protocol against some very advanced attacks. Most of the traffic will no longer be transported by the Simple Mail Transport Protocol (SMTP) [42]. Instead, the designers introduced a new protocol, which uses the Transport Layer Security protocol (TLS, [43]) to provide an additional layer of encryption.

2.2.5 Onion Routing

The public key operations which are performed at every hop on the path of a message in Chaumian mix networks are computationally expensive. So expensive, in fact, that it gets impossible to send near–real–time traffic through a mix network. If the mix is implemented with all precautions against the attacks listed in section 2.2.3, then messages have to be randomly delayed and reordered on their way, properties undesirable for fast, bidirectional communication.

An alternative, which sacrifices the unlinkability of messages, is "Onion Routing[4]", described by Syverson et al.[45]. The messages belonging to a *session* are distinguishable from those from other sessions by the nodes and all messages in a session follow the same path through the network. A special message type is used for session setup, in which the nodes on the path are informed about the *session identifier*, the secret *key* and the *next node* to use for that session. Only those initial setup messages are public key encrypted, the whole stream of messages in the session itself is encrypted with a symmetric cipher. This makes decryption computationally cheaper by several orders of magnitude while retaining the same (assumed) level of security.

[4] the term was introduced by Goldschlag in [44].

If the nodes $n_1, \ldots n_l$ are already informed by setup messages about the *next node*, the session ID id_i and the key k_i for each n_i, then a message m is encrypted by the sender as follows:

$$c_1 = (id_1, E_{k_1}(id_2, E_{k_2}(id_3, E_{k_3}(\ldots id_l, E_{k_l}(m)\ldots)))$$

Each node n_i on the path first looks up the *id* field in the table of open sessions, where the key k_i and the *next node* for that session are stored. It then strips away the *id* and decrypts the message with key k_i. The output is sent to the node indicated as *next node* for that session. This whole process can be imagined as "peeling off layers from an onion", hence the name of this class of protocols.

This protocol is a kind of faster analogy to Chaum's mixes. The nodes know the next and previous nodes for each session ID and thus can send data in both directions, decrypting upstream messages and encrypting downstream ones.

Crowds

A variant of Onion Routing is Rubin and Reiter's *Crowds* [46]. This protocol was designed as an HTTP anonymizing mechanism. A group of people (a *Crowd*) mutually relay their HTTP requests and the respective replies from the servers for each other. To make it harder for an observer to determine the origin of a request inside the crowd, a kind of Onion Routing is used. In contrast to Syverson et al.'s proposal, Reiter and Rubin do not use public key cryptography at all. On joining a Crowd, a host randomly chooses another host in the crowd as a relay for its subsequent HTTP requests. It sends a "setup" message to that host, which contains a path ID and a symmetric key. This relaying host registers the address and key for the path ID, so that subsequent messages carrying the ID will be en/decrypted and forwarded correctly. It then throws a biased coin (i.e. reads a fixed-sized number from a random source and compares it against a threshold). If the coin falls "tails" (i.e. the random number is smaller than the threshold), the path starting at the new host will terminate at this relay and requests from that path will be forwarded to the HTTP servers and their replies encrypted and sent down the path. If the coin falls "heads", the relay randomly chooses another host from the set of relays and sends a setup message, with new path ID and key, and registers the next host's address, the path ID and key as being the next hop in the path.

This implies that a host does not know where its request will be sent from. If a host inspects the traffic on the paths it is part of, there should be no hint as to the originator of the requests, since the previous host in any path could be just a relay for another one, as this host itself is.

2.2.6 Chaum's Dining Cryptographers Networks

The previous proposals listed here guarantee anonymity or unlinkability under assumptions, such as that a public key scheme is unbreakable, that only a

minority of the participants try jointly to expose the identity of a certain participant, and so on. Chaum's Dining Cryptographers (DC) protocol [47, 48] guarantees *unconditional* sender anonymity and unobservability as long as not more than $n-2$ out of n participants cooperate to trace the sender. His construction assumes a synchronous, reliable broadcast channel.

The name derives from an old resource management problem in operating systems design, the "Dining Philosophers Problem", in which the illustrative setting is a number of philosophers sitting at a round table. Chaum's original paper had a number of cryptographers sitting at a restaurant table after an expensive dinner. They are informed by the waiter that the bill has already been paid for by an anonymous party. They want to find out if somebody at the table paid or an external party paid for them. Since they respect the payer's right to anonymity, they want to do this without revealing the actual payer.

In a more abstract and general model, this problem is solved as follows: The participants are arranged in a 2-connected, one-component graph, so that every participant has at least two neighbors. To every neighbor, a participant has a secure channel (in Chaum's paper, the cryptographers hide behind the menu together with either their right or left neighbor).

Each secure channel is used to jointly generate a continuous stream of random bits. This "distributed flipping of coins" was discovered by Manual Blum in [49] (in Chaum's paper, the cryptographers flip a coin behind the menu).

As long as a participant does not want to send anything, she reads the generated bits from all her secure channels and XORs them (every cryptographer counts the "heads" or "tails" she sees). She sends the resulting bit to all participants on the broadcast channel (after all cryptographers have flipped coins with both their neighbors, everybody who saw an even number of "heads" raises their arm). If she wants to send a 1 bit, she inverts the XOR of the random bits, for a 0 bit she does not change it (the cryptographer who paid for the dinner raises her arm if she saw an odd number of "heads" or does not if she saw an even number).

To be able to receive messages, all participants listen on the broadcast channel, where everybody announces the XOR of their random bits. If nobody sends data, the XOR of all those broadcasted bits will be 0 (the number of raised arms will be even). This is because every random bit enters the XORing process twice — at both ends of a secure channel — and thus cancels itself out. If the result of the global XOR is 1, then somebody flipped a bit before broadcasting it.

If more than just one bit is to be sent, then it can happen that several people start sending at once; their message bits will be superimposed on the broadcast channel. There are several proposals to avoid superimposing of broadcast bits in DC networks. The simplest is the ALOHA protocol [50], familiar from the IEEE 802 line of protocols. In this, a fixed-format message header is prepended to every message and the messages have a fixed length. If any set of participants starts sending at the same, or almost the same time, they can see from the distorted or canceled header on the broadcast channel that somebody else is trying to send simultaneously. All senders then stop sending and wait for a random time–span. If a message is in the process of being sent after that,

they wait for the end of that message before they try again. For more elaborate schemes that use sophisticated reservation techniques, see [51].

The Dining Cryptographers unconditional sender-anonymous broadcast can be combined with conditional receiver-anonymous encrypted broadcast to produce an anonymous network that provides strong unlinkability.

Why are there no DC networks in use?

With all these properties, one wonders why there is no implementation of this protocol in actual use.

DC networks suffer from the problem of insider disruptions. Since senders are anonymous by design, anyone can interrupt or modify other participant's messages. There are quite a number of proposals in literature (see for example [52]) to subsequently find and expel the disrupter. All these add considerable complexity to implementations.

Another problem is the enormous amount of traffic that is caused, even when nobody sends anything ($O(n^2)$ message complexity under the assumption of a reliable broadcast channel).

The prerequisite of *reliable, synchronous* broadcasts makes the basic protocol impossible to implement on routed network protocols. A theoretically possible solution is to use a Byzantine Agreement protocol as the network layer. This suggestion regularly appears in literature, but a protocol for Byzantine Agreement over *asynchronous* channels was invented as late as 2001 by Shoup, Kursawe and Cachin [53]. The message complexity of this protocol would multiply the already enormous amount just mentioned[5].

2.3 Anti–Censorship Techniques

A different approach to free communication is to make it difficult for an adversary to censor messages on their way to the recipient, with the goal of achieving anonymous publishing. Anonymity of the sender is not always a requirement.

Early proposals were to spread copies over many legislatures so to make it difficult for oppressive regimes to order the removal of all copies. This method is still in use: when the source code for circumvention of DVD content scrambling was declared illegal in the United States, thousands of web–pages outside the U.S. started to provide the code.

More sophisticated techniques involve redundant storage combined with encryption. The much smaller decryption key is published after the message is stored, so that the dispersal of a message takes place before it becomes identifiable for the censor. Protocols with this property are for example Adam Back's EternityFS [54] and Ian Clark's Freenet [55]. To illustrate the principles, we describe Clark's proposal shortly.

[5]For every bit to be broadcasted, this protocol requires $3 \cdot n \cdot k$ (unreliable, asynchronous) broadcasts of cryptographic keys, where n is the number of participants and k the number of rounds. k remains small as long as the communication is undisturbed.

Freenet

The Freenet protocol provides a content dissemination mechanism like HTTP. In contrast to HTTP, the content is not tied to an IP address or host name. Freenet nodes query each other and store the replies in caches, much like HTTP's caching proxies, e.g., Squid [56]. Content is "injected" into Freenet and then is moved and duplicated towards the nodes with the most requests for it. On nodes which do not receive requests for it, it will be dropped eventually. To aid resource location, the content's immediate source is not deleted, but stored in a different table. Should a node request the expired content afterwards, this immediate source is queried.

That content is not tied to machines implies that it has a globally unique name. Freenet uses three kinds of names spaces:

Content Hash Keys: The introducer of the content c computes its hash value $h(c)$ and chooses a random encryption key k. This key is used to encrypt the content with a symmetric cipher. The content is stored and circulated under the name $h(c)$. Note that copies of the same content thus will have the same name. To retrieve the content, the hash and the encryption key are necessary. This guarantees integrity, because received and decrypted content can be checked against this hash. At the same time it allows different introducers to reveal the same content to receivers of their individual choice. This is because the receivers still need a decryption key, and each introducer can choose a different key and communicate it to whoever he or she chooses.

Keyword Signed Keys: The introducer takes a set of (possibly descriptive) keywords for the content and deterministically derives a public/secret key pair from the hash of the keywords. The content is signed with the secret part of the key. The public part of the key is hashed to produce the name under which the content will be stored.

The content is symmetrically encrypted with the hash of the keywords. This provides a kind of protection for the maintainers of Freenet nodes, because they can claim that they could not know the actual content stored on their machines.

To access the content, a user goes through the same steps and requests a file with the name derived from the public key.

Note that this does not provide message integrity or collision resistance, since the actual content never enters any of the computations.

Signed Subspace Keys: Introducers of content may create separate namespaces (subspaces), which are bound to public keys. To compute the signed subspace key of a file belonging to a given subspace, the public key of the names space is hashed and then XORed with the hash of a list of keywords. The outcome is hashed again and the resulting value — together with the public key of that subspace — will be the content's name inside Freenet.

Content is signed with the secret key for that particular subspace. After that, the content is encrypted with hash of the keywords as in the above section.

This guarantees that only the creator of a subspace can add new content to it, and allows retrieving parties to check the integrity of the received files.

The dynamic replication mechanism together with the encryption makes it difficult to remove specific messages from the network of nodes. If the adversary does not know the key to a message, she cannot request it to be removed from a node, since the maintainer herself can claim not to know which message it is and whether a copy of it resides on her node at all. If the adversary requests a known message by its key to find if it resides in the network, then this causes the message to be copied into the caches of potentially many nodes in the network.

One drawback of using only hash-derived unique ids for searching is that it makes fuzzy searches or keyword searches impossible. It is also impossible to generate a listing of all — or even a part of — the data stored in Freenet. There are several attempts to implement a searching in Freenet. They use Freenet itself to store tables of meta–data at announced locations in the network. An injector of data has to register its additions in those tables. This bears some similarity to early WWW search engines like Harvest [57], where content was actively registered in a hierarchical structure instead of being searched by "robots" as so–called search engines do.

2.4 Receiver–Anonymous Protocols over unreliable Media

If a protocol has to assure full receiver anonymity (against a global observer), then for every sent message there must be a message delivered to every member of the anonymity set. To elucidate this, assume that l messages were sent in a period of time by a set of participants S and a certain set of participants R did only receive $k < l$ messages. Then the observer can conclude that there are at least $l - k$ members of R that were not communicating with a sender in S (proof by pigeon–holing). This violates the strong requirement that every party be as likely a receiver as any other.

If for some reason delaying and reordering of messages is undesirable, then this requirement enforces some variant of broadcasting to all members of the anonymity set.

Should the broadcast medium in this case be unreliable, then caution must be taken when designing mechanisms for re–transmission of lost messages.

We will shortly describe multicast channels in IPv4 and IPv6 and explain why they are interesting for designers of anonymity protocols. We will however point out why extra caution is necessary if receiver anonymity against an adaptive local attacker is required.

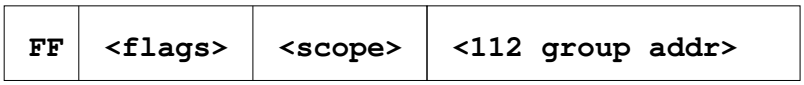

Figure 2.2: IPv6 multicast address

2.4.1 Multicast as seemingly attractive broadcast channel

In multicast, the destination address of a datagram is of a special form, in IPv4 it is an address from the range 224.0.0.0 through 239.255.255.255, in IPv6 the address starts with the special `FF` prefix [58]. The address designates a *group* of machines.

The multicast routing protocol uses spanning trees (see for example [59]) to avoid routing loops and unnecessary duplication of packets along the path to their respective destinations. The root of the tree will lie in the multicast backbone, if the group has wide–spread subscribers.

To join an IPv4 multicast group, a process sends an Internet Group Management Protocol "join" message [60] for that group. The local router checks if it is already a member of that group. If not, the router itself sends a join message to the next router on the path to the MBONE, a special backbone network for multicast traffic. How multicast traffic will be routed on the IPv6 backbone is not quite clear yet.

Multicast datagrams carry information as how far they are to be propagated through the spanning tree. In IPv4, the Time To Live (TTL) field in the IP header specifies how many routers a datagram may traverse before it is discarded, so a multicast datagram with TTL of 1 will never leave the local subnet, whereas a TTL of 255 (the maximum) should guarantee that the datagram is forwarded and duplicated through the whole tree associated with the group.

In IPv6, different prefixes ("scopes") signify how far a datagram is to be forwarded. The prefixes as specified in [61] consist of four parts: a constant prefix `FF` at the beginning, followed by four bits for flags, four bits for the scope and 112 bits for the multicast group identifier.

`flags` The lowest of the four flag bits indicates whether the multicast address is "well–known" (`flag = 0000`), i.e. is registered with the Internet Assigned Numbers Authority (IANA), or not (`flag = 0001`). The other three bits are reserved.

`scope` limits the scope of the multicast group. Possible values are:

scope bits	hex	name of scope	reach of scope
0001	1	node local scope	The datagram never leaves the hosts.
0010	2	link local scope	The datagram never leaves the local network.
0101	5	site local scope	The datagram never leaves the an administrative domain "site".
1000	8	site local scope	The datagram never leaves the an administrative domain "organization".
1110	E	global scope	The datagram is propagated until the hop-count header of the IPv6 datagram reaches 0.

For example, the address FF0E::101 is allocated by the IANA for all Network Time Protocol [62, 63] servers on the Internet.

2.4.2 Multicast and receiver–anonymity

If a multicast group is used as broadcast channel for an anonymizing protocol, this transport over a spanning tree's nodes produces an interesting property as side effect: the higher up in the spanning tree an observer is positioned, the better the actual destinations of the packets are obscured from her. This is contrary to the intuitive assumption, *viz.* that an observer on the backbone should be in the best possible position to inspect traffic.

To the author's knowledge there has been only one proposal of using multicast groups for anonymity protocols, by Shields [64]. Shields assumes that participants can fake source addresses and uses this to achieve sender–anonymity with respect to a local observer on the multicast backbone.

Note that the concept of a passive global observer is totally reasonable in the multicast setting. The observer can join the multicast group from anywhere and will see all the messages.

One common technique to make correlation of messages harder is sending dummy messages. These are just chunks of randomness, with correct headers attached to disguise them as real messages. Since the output of a cipher should not be distinguishable from randomness, an observer cannot tell dummy messages from real ones. But protocols employing dummy messages do not scale well in a multicast setup. If the minimum bandwidth of all n participants is k messages per time unit t, then at least one party's connection will be saturated as soon as k participants send real or dummy messages. Dummy messages thus reduce the *overall* bandwidth, whereas in setups such as mix networks they only reduce the bandwidth of certain paths. If the number of participants is not fixed (which will most often be the case) then the algorithm for sending dummy messages cannot be fine–tuned to accord to the channel's minimum bandwidth.

To avoid flooding of parties on thin lines, all participants would either

1. need to be able to detect each other's dummy messages and regulate their own output accordingly.

2. or somehow globally announce their available bandwidth.

In case 1 an attacker need only to join the channel to detect and discard dummy messages, thus rendering them useless. In case 2 the invisibility of the receivers in the multicast setting is violated, because they would have to *send* something.

After the exclusion of dummy messages as protocol features, another problem protocol designers face is the unreliability of multicast channels. The datagram type for multicast is the User Datagram Protocol (UDP, [14]), which does not provide any inherent mechanisms for reliable transport. So protocols employing multicast must make their own provisions to guarantee delivery.

Achieving Reliability

Methods to provide such guarantees are

1. positive acknowledgements as in TCP [15].

2. negative acknowledgements as in the High–level Data Link Control protocol (HDLC [65]).

Positive acknowledgements are sent on receipt of a message. Negative acknowledgements are sent automatically after a pre–defined period (the *time–out*) passed without receipt of the expected message.

For the rest of the chapter we assume that there are no dummy messages. For analysis we further assume that the data sent by the participants most often does not fit into a single message, but has to be split into several messages. Our attacks are not hampered by replay protection, but without replay protection adaptive attacks become significantly easier. We assume that the attacker can guess a lower bound n on the number of participants. This is not unreasonable, the attacker could for instance simply count the observed IP source addresses on the channel for some time.

A Formal Definition of Unlinkability

To be able to reason about general protocols without their actual specification, we use probabilities to describe unlinkability in arbitrary protocols.

We follow the formal description suggested by Steinbrecher and Köpsell in [66]. Here, the *a priori* probability $P_t(i \sim j)$ that two items of interest i and j are related should be equal to the *a posteriori* probability $P_{t+1}(i \sim j)$ after observation. This is a strong requirement, because it means that *nothing* can be learned at all by observing the flow of messages.

The item of interest in our case are the sending and receiving parties of the broadcast channel, the relation $i \sim j$ is the fact of i having sent a sequence of messages to j.

In the broadcast setting, the upper bound of the *a priori* probability that i is the intended receiver of messages send by j is

$$P_0(i \sim j) = \frac{1}{n-1} \ .$$

No positive acknowledgements in the adaptive attacker scenario

If the assumed attacker is adaptive and local as defined in section 2.1.1, then positive acknowledgements violate the receiver–anonymity requirement.

Here is how an adaptive attacker can compromise receiver–anonymity:

1. The attacker wants to learn something about the receiver/s of messages sent by party A.

2. She takes control over the leaf router in A's subnet.

3. For every message $m_i, \qquad i \in \{1, \ldots, k\}$ that A sends, she notes down the set of senders S_i of other messages in the period till A sends the next message.

If a participant was the intended receiver of A's messages[6] then she must have sent an acknowledgement after each message. From this follows that while $P_0(i \sim j) = \frac{1}{n-1}$, for $j \in \bigcap_{t \in \{1,\ldots,k\}} S_t$, after k messages from A,

$$P_k(i \sim j) = \frac{1}{|\bigcap_{t \in \{1,\ldots,k\}} S_t|}$$

which will be greater than $P_0(i \sim j)$ unless every participant known to the observer sent a message after each of A's messages.

So by the strong requirement set in section 2.4.2, all broadcast–based protocols fail to provide receiver anonymity if they

1. do not use dummy messages.

2. try to achieve reliability of delivery by automatic positive acknowledgements.

No negative acknowledgements, either

If negative acknowledgements are to be used, the number of messages inside a session must be announced at the start of the session. Else the receiving party would not know whether to expect further messages or not. In addition to that, a time–out period T must be defined, after which negative acknowledgements are sent automatically. Not using a time–out implies that messages are never de–queued by the sender, since there could always be a re–transmission request later. Here is how an adaptive attacker can compro-

[6] and there must have been one, or else those would have been dummy messages.

mise receiver–anonymity:

1. The attacker wants to learn something about the receiver/s of messages sent by party A.
2. She takes control of the leaf router in A's subnet.
3. She blocks all messages in the channel from entering A's subnet.
4. She waits for at least $2 \cdot T$ periods of silence from A.
5. If A sends a message, she removes the block from step 3.
6. As long as A does not pause for at least $2 \cdot T$ periods, she discards every second message A sends. She also delays the non-discarded messages to the next T period. She notes down the set of senders S_i which send messages after T periods following the discarded messages.

Steps 3 and 4 are to ensure that all of A's sessions are closed and the next message is the initial one of a sequence. After receiving A's initial message the receiver knows how many messages to expect. So for every discarded message, the receiver will issue a negative acknowledgement after the time–out. From this follows again that while $P_0(i \sim j) = \frac{1}{n-1}$, for $j \in \bigcap_{t \in \{1,\ldots,k\}} S_t$, after k messages from A,

$$P_k(i \sim j) = \frac{1}{|\bigcap_{t \in \{1,\ldots,k\}} S_t|}$$

which will be greater than $P_0(i \sim j)$ unless every participant known to the observer sent a message after every second of A's messages.

By the strong requirement set in section 2.4.2, all broadcast–based protocols fail to provide receiver anonymity also, if they

1. do not use dummy messages.
2. try to achieve reliability of delivery by automatic negative acknowledgements after a pre–defined time–out.

Inclusion of cover traffic in broadcast protocols seems a possible way out of this dilemma. How to leverage cover traffic without seriously affecting the overall bandwidth is however a problem in itself if the number of participants cannot be fixed.

Tunneling reliable protocols in unreliable ones

If no provisions are made for reliability in the design of a broadcast–based protocol, then users should be made aware of that. For if they chose to tunnel e.g. TCP through the protocol, the SYN/ACK mechanism of TCP will open the communication to the attacks just described.

2.5 Results of this chapter

This chapter gave an introduction to anonymity providing protocols. After defining necessary terms, various proposed and implemented systems were presented, most notably Chaumian Mixes and derivatives thereof. We also described an example of an anti–censureship protocol which tries to guarantee the availability of messages in face of an active adversary. Since broadcast combined with public key encryption seems an interesting alternative to Chaumian mixes, given that multicast is available in IPv6, we examined how anonymous communication can be accomplished over such unreliable channels. We proved that the standard means of making lossy channels reliable cannot be applied in this scenario without losing the sender–receiver unlinkability as defined by Köpsell and Steinbrecher.

Chapter 3

Covert Channels in the TCP/IP Stack

3.1 Overview

This chapter examines the potential for tunneling data in well–established protocols. First we will explain the concept of tunneling. Covert channels, subliminal channels and related concepts will be our next subjects. We will present arguments for a broader definition of covert channels, which includes network connections, to adapt the concept to modern operating systems and environments. This is followed by an exhaustive classification of covert channels in TCP/IP protocols.

In the presence of filters, parties wanting to communicate cannot freely choose the communication channels for new protocols. Binding a server to a TCP port number outside the small set of: 20 (ftp-data), 21 (ftp), 22 (ssh), 23 (telnet), 25 (SMTP), 53 (DNS), 80 (WWW), 110 (pop3) and 443 (https), for example, will cause unreachability of the server in most network setups. Using these services openly would make the act of communication obvious. For the reasons given in the introduction, we want to avoid this.

To allow unobserved communication, without fear of repercussions, we propose to embed the messages in unsuspicious, widely–used protocols. This has to be done in a way such that an observer should not notice the embedded messages, or only after analyzing huge amounts of traffic.

3.2 Tunneling

For a network protocol that has separated layers, tunneling is defined as injecting network packets into the stack of layers after they have already passed some of the layers below the point of injection.

Example: IP over HTTP

The `httptunnel` utility by Lars Brinkhoff [67] allows arbitrary applications to communicate by HTTP messages. This is typically used to connect two `pppd`

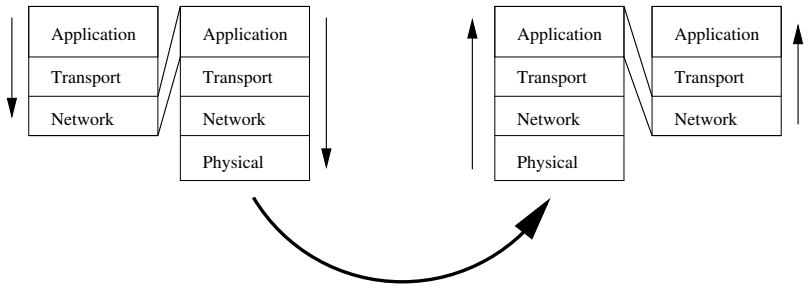

Figure 3.1: Tunneling

processes on different sides of a firewall. `pppd` in turn creates a new network device which provides a full IP stack. If an application sends data through this new device, the data goes through a complete TCP/IP stack before it is repackaged as HTTP application data and send once more through the TCP/IP stack of another device. On the other end of the connection, the HTTP data is parsed and reassembled into the link–layer protocol PPP [11]. This data is handed over to `pppd` which sends it upwards through the TCP/IP stack.

3.3 Covert and Subliminal Channels and related topics

The means of data transportation, *channels*, for our purposes, should have the property that they are hard to detect and/or hard to block for a variety of adversaries. Such channels have been studied in different contexts, and we will present the concepts of these and how they are related.

3.3.1 Covert Channels

The notion of Covert Channels was introduced by Butler Lampson [68] in 1973 in the context of secure operating systems. He demonstrated several ways of communication between processes in different, presumably secured compartments. Covert channel analysis has since been a topic in design of secure operating systems, see for example the *Navy Handbook for the Computer Security Certification of Trusted Systems* [69], the Common Criteria Evaluation and Validation Scheme [70], or [71].

The definitions of covert channels in literature differ widely, from

> *system behaviors that surprise the designer of the system.*

by Marv Schaefer, or

> *Covert channels are those that "use entities not normally viewed as data objects to transfer information from one subject to another".* [72]

to highly formal ones, see for example [73]. Covert channels can be categorized in two classes, timing channels and storage channels. In timing channels, delays on other channels are artificially caused and measured to transfer

information. Storage channels modify information in other channels to transmit messages.

3.3.2 Subliminal Channels

Subliminal channels (sometimes called "sublime channels", although *sublime* and *subliminal* have very different meanings) were discovered by Gustavus J. Simmons in 1983 [74, 75, 76] in digital signature protocols. He pointed out a design flaw in the protocols intended for the enforcement of the Strategic Arms Limitation Treaty (a nuclear weapons control treaty). He showed that the signing party could embed information in the signatures un-detectably, although the messages themselves where extremely closely scrutinized. He later demonstrated subliminal channels in the Digital Signature Algorithm (DSA, [77]) and other probabilistic signature schemes. A number of cryptographic authentication protocols exhibit subliminal channels, e.g., most zero knowledge identification protocols such as Feige–Fiat–Shamir [78]. The specification of the (RSA based) Provable Secure Signature Scheme in the PKCS#1 standard, version 2.0 explicitly states the presence of a 160 bit subliminal channel.

The main difference between covert and subliminal channels is that subliminal channels *cannot* be detected because they are a side–effect of the security of a cryptographic scheme.

Subliminal channels in signature schemes transport data in fields reserved for authentication data when probabilistic schemes are used. This goes undetected because of the non–uniqueness property of such signatures: there is a (presumably) random parameter involved in the signing algorithm which can be chosen by the signer and in which the subliminal message can be embedded. The recipient will only be able to recover this subliminal message if he or she has access to the private key of the signer[1].

As with covert channels, a strict formal definition of subliminal channels is very difficult, as remarked by Y. Desmedt ([81] and [82]). This is — at least partially — because the classical notion of communication channels as defined by Shannon is not applicable in this context. Consider for example the *entropy* of a secret key belonging to a known public key, which by classical measure should be zero (there is a one–to–one relationship, i.e. every bit in the secret key is determined by the public key). But in the following example it will become clear that knowing the secret key opens a channel of non-negligible bandwidth.

Example: Subliminal Channel in the Digital Signature Algorithm

The Digital Signature Algorithm (DSA), also known as Digital Signature Scheme (DSS) [77], was designed by the US National Institute of Standards and Technology (NIST) to provide a signing mechanism which uses public keys, but does not fall under the export restrictions for encryption devices.

[1] There is an advanced scheme by Anderson et al. [80], where only part of the secret key needs to be shared at the penalty of reduced bandwidth.

DSA was specifically designed not to allow public key encryption, in contrast to the RSA [83] or ElGamal [84] algorithms.

DSA is widely used, for example in DNSSEC [85], X.509 [86] and PGP [40]. We first describe the algorithm and then show how to embed a 160 bit subliminal channel into it.

The secret key in the DSA scheme is a 160 bit number x. To generate a public key, the following computations are executed:

1. Find a prime number p of about 1024 bits such that $p-1$ is divisible by a 160-bit prime q.
2. Find a generator g of the cyclic subgroup of order q in $\mathbb{F}_p$.
3. Compute $y = g^x$.

The public key is the vector (p, q, g, y).

To generate a signature for a message m, one does the following:

1. Compute the hash $h(m)$ where h is the Secure Hash Algorithm [87].
2. Randomly and securely choose a number $k \in \{2, \ldots, (q-1)\}$.
3. Compute $r = (g^k \mod p) \mod q$.
4. Compute $k^{-1} \mod q$.
5. Compute $s = k^{-1}(h(m) + x \cdot r) \mod q$.

The signature for m is the pair (r, s). The perceived advantage of DSA over RSA is that the signature is shorter (320 bits instead of 1024 or more) and signature verification is faster, because some of the operations are done in a group of order q, which is much smaller than the group of units in $\mathbb{Z}_n$ as in RSA, where n is at least 1024 bits long.

To verify a signature (r, s) on a given message m, one computes:

$$\begin{aligned} w &= s^{-1} && \mod q \\ u_1 &= w \cdot h(m) && \mod q \\ u_2 &= rw && \mod q \\ v &= (g^{u_1} y^{u_2} \mod p) && \mod q \end{aligned}$$

and accepts the signature only if $v = r$.

Note that k in step 2 must not be predictable in any way. If k were guessable for an attacker who knows m and the signature (r, s), then he could compute:

$$\begin{aligned} v' &= s \cdot k && \mod q \\ &= k \cdot k^{-1}(h(m) + x \cdot r) && \mod q \\ &= (h(m) + x \cdot r) && \mod q \\ w' &= v' - h(m) && \mod q \\ &= x \cdot r && \mod q \\ x &= w \cdot r^{-1} && \mod q \end{aligned}$$

and thus obtain the secret key. If the signer S shares his secret key x with another entity P, then S can use k in step 2 of the signing procedure to send 160 bits to P subliminally in a signed message (m, r, s). This works because only P and S know x and can compute

$$\begin{aligned} u &= s \cdot (h(m) + x \cdot r)^{-1} && \mod q \\ &= k^{-1}(h(m) + x \cdot r) \cdot (h(m) + x \cdot r)^{-1} && \mod q \\ &= k^{-1} && \mod q \\ k &= u^{-1} && \mod q \end{aligned}$$

Note that this contradicts the design requirements for DSA, but only partially so, because the channel implements symmetric encryption, not public key encryption.

3.3.3 Related Topics

Data hiding has been studied in other contexts besides the two named above. In recent years a lot of work was done in the direction of what might be considered "copy-protection post hoc", but borrows its name from a proven anti–counterfeiting technique, viz. *Watermarking*. A very different motivation for data hiding is to send data "under the radar" of possibly oppressive administrations. The term commonly used for these techniques is *steganography*. This is what we are interested in here.

Steganography

In steganography, encrypted data is hidden in the communication payload. This is possible if the payload, the *cover-text*, contains enough entropy to allow unsuspicious modification. Content formats that typically exhibit this property are images in various compression formats, digitally encoded sound and motion pictures in various encodings. These have the additional advantage of being relatively bulky, so that even with small embedded bandwidth considerable amounts of data can be hidden in them.

Advanced steganographic techniques (for example Outguess [88]) typically search for portions of the cover-text that have high entropy (i.e. look almost random) and do not contribute too highly to the large–scale appearance of the cover-text. The data to be hidden is then embedded in said portions while trying to maintain the statistical properties of those regions.

Note that encryption is used here not only to make the data unreadable in case it should be extracted by an observer. In addition to that, proper encryption transforms data so that the output cannot be distinguished from randomness, which is just what the encrypted text will be used to replace.

For example, in a GIF–encoded [89] image, the least significant bits (LSBs) of each pixel are steganographically usable, but not all are suitable in all circumstances. Caution should be taken so that a steganographic tool does not try to modify the LSBs of pixels inside a totally monochromatic region of the image, for instance.

Watermarking

Watermarking is closely related to steganography. Whereas in steganography the observer should not be able to detect the embedded information in the cover text, in watermarking it is assumed that the observer already knows about the presence of this information, but should be prevented from removing it from the cover text. Embedding in watermarking schemes thus has to be resistant against change of encoding, further compression, slight distortion, etc. Not the presence of embedded information is obscured, but the actual places where it is embedded. The main application is to mark pieces of information per purchaser, so that copies of said information can later be traced back to their original purchaser, for purposes of litigation.

At the time of writing watermarking schemes have been broken by the score in practice (see for example Markus Kuhn's Stirmark [90]), and it remains to be seen if their planned application in court will yield to the expectations of their proponents.

Chaffing and Winnowing

When it became clear that the U.S. export restrictions would not apply to authentication–only software, Ron Rivest demonstrated in [91] how to communicate securely through authenticated channels, even if no encryption is employed. The idea behind his method is that only the intended receiver will be able to distinguish fake datagrams from authentic ones. So for each datagram, the sender transmits a large amount of alternative datagrams, which carry incorrect Message Authentication Codes. An observer sees many variants (which could be combined in exponentially many ways, multiplying with the number of datagrams in a session) and cannot decide which will be discarded by the receiver and which are part of the actual communication.

3.4 New perspectives on channels

Research in the fields just presented, i.e. covert channels, subliminal channels and steganography, is done by mostly disjoint communities. Covert channels are examined by operation systems researchers in the very specialized field of multilevel security systems, e.g., those following the Bell-LaPadula design [92]. The design of systems so that they specifically do or do not contain subliminal channels is looked into by few cryptographers. Steganographic techniques are mostly developed with a far less theoretical motivation, with less formal proofs based on statistics. Watermarking is the subject of growing "market–driven" research and became almost totally disjoint from steganographic research for economic reasons. We would like to combine results from these fields to examine the potential channels inside existing Internet protocols. Suitable channels will later be used to construct hidden communication networks in Chapter 5.

Since networking has become a standard feature in all major operating systems (even to the point of dependence on remote procedure calls or distributed

components), the concept of covert channels will be broadened in our context to include communication outside the borders of the host's operating system. We will define a notion of (broadband) covert channels in the context of networked computers. We follow the example set in the Common Criteria Evaluation and Validation Scheme [93] and will not describe channels with very low bandwidth (≤ 100 b/s). Because timing channels of noteworthy capacity are hard to establish on best–effort packet–switching networks, we will ignore them completely.

The policy–enforcing or censuring party are now filters (also called "middle–boxes" in recent IETF publications) that try to regulate and audit the data stream. Their inspection can be avoided if the channel is subliminal or steganographic techniques can be employed. In our notion, covert channels can be classified with respect to the filters' possibilities to block them in real–time or detect them in retrospect by inspection of logs. Note that *overt* encrypted communication does by definition not constitute a covert channel.

3.5 Covert Channels inside TCP/IP Protocols

Protocols on all layers of the TCP/IP protocol stack define fields which can or even must contain random–looking data, depending on the observer's scope. Randomness is introduced for various reasons:

Uniqueness If the payload data of the protocol is to be referred to later on, the data has to be tagged with a unique ID. This is the case with the `MessageID` header in e–mail messages, as defined in RFC 822 [24]. If the scope of possible references is broad, as in RFC 822[2], then it is advisable to assure uniqueness by at least adding randomness to the field. If the scope is local, as in messages generated from database entries (one of the projected applications of XML), a simple counter can be used. If the database is large, even counters will look random to an observer (this is the case with the Unified Resource Locators (URLs) generated by `amazon.com`'s CGI scripts). Unique, random tags can be found in the specifications of popular application data formats, e.g., Microsoft's Rich Text Format [94] as used by Microsoft Word and other word processors.

Freshness If a protocol has to assure that a received message is part of a recently started exchange of messages with one participant, fields with random data are employed, sometimes called *cookies*. The answer to a TCP SYN datagram (session initiation) in the TCP protocol, for example, has to echo the Initial Sequence Number from the first datagram [15]. If those numbers were chosen in a predictable fashion, then certain attacks become feasible (see for example [95]).

Message Authentication After two parties have agreed on a shared secret, they can authenticate their messages by appending Message Authenti-

[2] An e–mail should be uniquely referable for anyone, at any time.

cation Codes (MAC). Those MACs are random data, unverifiable and unforgeable for an outsider.

Challenge–Response Authentication Since passwords should not be sent in the clear, the common practice for a verifier V is to send a random string c (the challenge), which the prover P "encrypts" with the password and sends the result r (the response) back. V encrypts the random string with P's password and compares the result with the received r. If they match, then P knows the password. In this exchange, both parties send (statistically) random strings. This type of authentication is employed in protocols such as SASL [96], HTTP [97] and Microsoft's Shared Message Block [98] protocol.
It should be noted that an active adversary could modify the challenge c. If the response r contains only the payload of the covert channel, the adversary will detect the channel. This is because the verifier should deny access to whatever is protected by the authentication protocol, since the response does not match the challenge. Detection can however be thwarted by using only a part of the response for hidden data and filling the rest with the respective parts of a valid response. If for example 112 of 128 bits in a challenge-response scheme are used as covert channel, then the probability for an active adversary detecting the channel with one attempt is 2^{-16}. The security of the authentication scheme decreases linearly with rising bandwidth of the channel.

3.5.1 Classification of Channels

We introduce two orthogonal scales for covert channels. The first expresses the visibility or invisibility of embedded data. A channel is called *obvious*, if close scrutinity of a single data unit of the respective protocol layer allows discovery, and *safe*, if only analysis of many units can lead to discovery, if at all. If the adversary is able to modify traffic and observe the replies to the modified messages, some of the safe channels will become detectable, however.

The second scale expresses the theoretical possibility or impossibility of blocking the channel without breaking the respective protocol. We will call a channel *blockable* if it is possible to substitute the embedded data in real–time, and *unblockable* if the design of the protocol prohibits such substitutions. Note that in the classification we will call channels blockable even if all *practical* measures might fail to eradicate the channel.

3.5.2 Blocking of selected Channels

If filters are to permit the overt communication over TCP/IP based protocols, but at the same time to close at least some of the covert channels in those protocols, then the filters have to substitute the contents of protocol fields of data units on their way out of the filtered network, and to substitute those values back on the way in, if necessary. Below the application layer, the filtering is partially achieved already as a side–effect of Network Address Translation

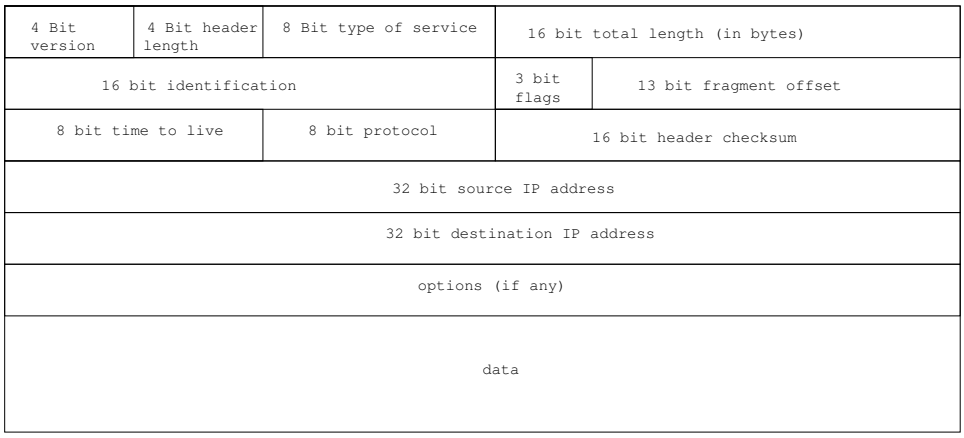

Figure 3.2: IPv4 Header

(see section 1.1). Building filters that modify protocol fields on the application layer efficiently and in near–real–time without breaking the protocol all–together, is a hard problem as we will show below.

We now examine protocol fields on different layers of the TCP/IP protocol stack for their suitability as channels for covert communication:

Physical Layer Although a number of techniques are available for covert channels in standard telecommunication media (e.g. spread spectrum modulation), we will not discuss protocols on this layer, since their signals will not travel farther than the next router or switch.

Link Layer Covert channels in the link layer will be omitted for the same reason.

Network Layer: IPv4

1. **IPv4 Identification** The IPv4 header contains a 16–bit identification number, which is used to identify the fragments of a packet at the receiver. The IETF standard suggests that it is to be implemented as a per-host counter, but some implementations use random values instead. For an observer outside the local network, a counter is not easily distinguishable from random values, since the datagram visible to such an observer might constitute only a negligible part of the hosts IPv4 traffic (e.g., if parts of the file system are accessed over network file-systems).
   Classification: The IPv4 identifier is *safe* and *blockable*. Firewalls such as OpenBSD's `pf` [99] supply methods to substitute the identification fields on the gateway.
2. **IPv4 Options** The IPv4 header may have options attached. The option with the most space is the IPv4 Security option [100]. This was intended by Department of Defence developers in the early '80s to allow labeling of data streams so that multi–level security

systems [101] could direct the stream to the appropriate compartment[3]. Other IPv4 options are loose/strict source routing and the "Don't fragment" and "Is fragmented" flags. The bandwidth of the flags is too small to be considered here.
Classification: IPv4 options are *obvious* and *blockable*. Standard firewall and NAT configurations strip away the source routing options, an act which — in a standard setting — even optimizes the path of the packets without affecting the protocol at all. No modern operating system known to the author has a programming interface for processing IPv4 security options on incoming datagrams, so simply dropping those bytes will not affect the higher level protocols.

3. **IPsec AH HMACs** The IP Security Standard defines Authentication Headers (AH) [102]. In AH mode packets are not encrypted, their content, including some header fields like the addresses and ports, is protected against tampering by a hashed message authentication code (HMAC). To generate and subsequently check these codes, both communicating parties have to share a secret s. The security of this construct relies (under standard cryptographic assumptions) only on the secrecy of s.
Classification: HMACs are *safe* and *unblockable*.

4. **IPsec ESP** The specification of the Encapsulated Security Payload in IP Security allows to use the NULL cipher for encryption. If run in this mode, the payload is still authenticated as in AH mode described above. The authentication mechanism is the same, but only the body of the IPv4 packet is authenticated. This mode of operation is sometimes used instead of AH mode, because it does not break in the presence of NAT.
Classification: Like AH HMACs, the ESP NULL encryption with authentication is *safe* and *unblockable*.

Network Layer: IPv6 The successor of IPv4 is IPv6, standardized in a block of RFCs starting from RFC 2460 [103]. Although many operating systems support IPv6, deployment is low at the time of writing, and not all features have been tested thoroughly. The channels we describe in this section are available only if deployment follows the standards closely. Note that IPv6 uses header chaining, where optional headers and transport layer headers are daisy–chained to the IPv6 header through the `next header` fields in each successive header.

1. **IPv6 source address** The IPv6 source address for hosts is built automatically from the host's subnet's 64 bit prefix and the 48 bit hardware address of the network interface card, i.e. the hardware address is split into halves of 24 bits and `0xFFFE` is inserted in between. A neighborhood solicitation message is sent on a multicast channel to check if the address is already taken by some other host. RFC

[3]This assumed that IPv4 traffic cannot be altered on the path and is therefore a heavily flawed concept.

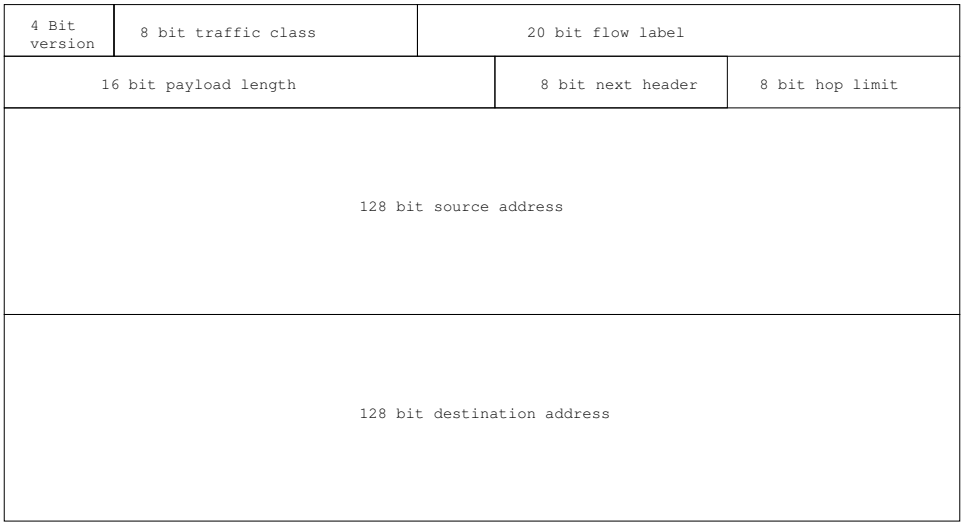

Figure 3.3: IPv6 Header

3041 [104] specifies a privacy extension to this mechanism. This allows a hosts to create temporary source addresses. This can be exploited to embed 8 bytes of data into the source address by setting the lower 64 bits accordingly.
Classification: Embedding in source addresses is *safe* and *unblockable*. Note that both the non–obviousness and unblockability depend on RFC 3041 being supported by manufacturers of routers.

2. **IPv6 flow label address** The flow label is a header field planned to facilitate routing based on flows or virtual circuits. The standard [103] prescribes the field to be random, so that it can be used as key for table lookups in the routers directly. This opens a covert channel of 20 bits.
Classification: Embedding in flow labels is *safe* and *unblockable*. Changing the flow label would break the virtual circuit mechanism and defy the purpose of the protocol field.

3. **IPv6 Destination Option header** This is a header of dubious purpose. The RFCs do not indicate any application, but this type of header can carry up to 256 bytes of arbitrary data. To the author's knowledge there is no implementation of the IPv6 stack that honors this header type. It allows for 256 bytes to be piggy-backed on every datagram.
Classification: Embedding in the destination option header is *obvious* and *blockable*.

4. **IPv6 Fragment Header Identification** IPv6 routers do not fragment datagrams, fragmentation is performed by the sender. Fragmented datagrams carry a special fragment header. The fragment IDs provide us with a covert channel like the IPv4 fragment identification,

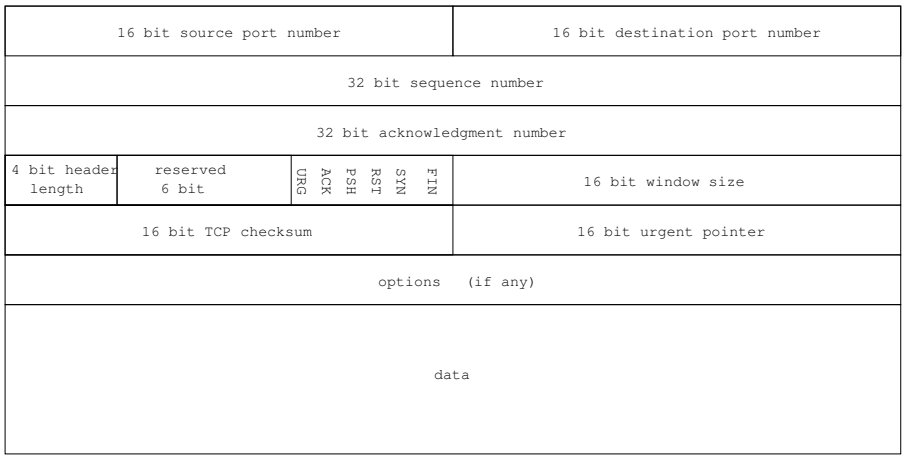

Figure 3.4: TCP Header

with the advantage that in IPv6 the IDs are twice as long.
Classification: The IPv6 identifier is *safe* and *blockable*. The same reasoning applies as for IPv4 fragment IDs.

Transport Layer

1. **ICMP Echo Body** RFC 792 [105] specifying the Internet Control Message Protocol states: "The data received in the echo message must be returned in the echo reply message." This implies that an arbitrary amount of data minus the IP and ICMP header length can be included in a single `ping` datagram. The fact that ICMP echo requests and replies are generally not blocked at firewalls has given rise to a number of tools which use this as a covert channel, for a detailed discussion of these, see [106].
   Classification: The ICMP Echo body as a channel is *obvious* and *blockable*. Though the Unix command `ping(8)` has options to fill the body of an Echo request with specified data, this is not crucial for the functionality of the protocol, that is, to discover if a host is up or not.
2. **TCP Initial Sequence Numbers** The Transmission Control Protocol [15] provides a reliable transport mechanism. To guarantee delivery, the octets are counted and the receipt is acknowledged by an answering datagram including the number of the last received octet. The first datagram in a TCP session sets a 32–bit initial value for the octet counter, called Initial Sequence Number. If this number is predictable, several so–called TCP Hijacking attacks are possible by adversaries that do not even need to tap the session, see for example the CERT advisory CA-2001-09 [95] and [107]. The various BSD and Linux implementations of TCP are secure with respect to

this and use random numbers as Initial Sequence Numbers.
Classification: TCP ISNs are *safe* and *blockable*. For example, the OpenBSD firewall and NAT code `pf(8)` [99] substitutes sequence numbers by random ones automatically for both directions in real–time.

3. **TCP or UDP Source Ports** To distinguish different programs on one machine all connecting to a single service on another, TCP and UDP [14] identify sessions by source and destination addresses and source and destination port numbers. The set of those four numbers is unique for each session. The source port is usually assigned by the operating system, but the `socket` Application Programming Interface allows processes to request specified source ports. Again, using predictable source ports might enable an attacker to hijack the connection. For this reason, the OpenBSD [108] implementation uses a strong random number generator for creation of source port numbers.
Classification: Port numbers are *safe* and *blockable*. NATs necessarily substitute the port numbers.

4. **TCP Urgent Pointers** The original design of TCP included an additional data stream for "urgent" data. For this, a special flag in the headers and a 16–bit data field (the Urgent Pointer) is reserved. If the flag is set, the application is informed by the operating system and the Urgent Pointer is handed to be processed. To our best knowledge, the only application that ever used this feature was `telnet`, which set the Urgent flag if the interrupt key was pressed on the client side. The Urgent Pointer then contained the position of this keystroke in the datagram.
Classification URG Pointers are *obvious* and *blockable*.

5. **TCP Window Size** To save time on waiting for acknowledgements (ACKs), TCP sends ACKs for whole blocks (*windows*) of octets, not for datagrams. The size of these windows is expanded or shrunk depending on the quality of the connection (the perceived quality, to be exact. Why the current protocol is easily deceived about that quality see [109]).
Classification The TCP Window Size is *obvious* and *blockable*. The changes of the Window Size correlate to observable factors as the Maximum Segment Size and recently received ACKs from the receiver. If some kind of traffic conditioning is used on any of the intermediate routers, setting the Windows Size to random values will break the TCP session eventually.

6. **TCP Timestamp Option** To get better estimates on the Round-Trip-Time, RFC 1323 [110] introduced the TCP timestamp option. If this option is used, every acknowledgement (ACK) carries two timestamps. One is the current time on the machine sending the ACK (the TSval field), and the other (the TSecr field) is the freshest timestamp received in the TSval field from the other machine. Both timestamps

are 32 bits wide. Data can be embedded by changing the timestamp of the sending machine, or both timestamps.
Classification Embedding data in both timestamp fields is *obvious*. Embedding only a few bits in the lower bits of the TSval field is *safe*. Both methods are *blockable*, because the timestamp option is not mandatory and can be removed by middleboxes.

Application Layer We examine only the most widely used applications.

1. **Simple Mail Transport Protocol (SMTP)** SMTP [42, 24] is one of the the most popular application protocols on the Internet. The headers defined in [24] and [111] contain the following fields which can be used as channels
   (a) **MessId** The message identification header [24] has to be globally unique for all time. This is a strong requirement. Most mail user agents achieve this by building the MessId from the fully qualified host-name of the machine (globally unique by RFC 1035 [112]), the user-name (unique per system), the current time up to the second (unique after a second) and a few bits of randomness. This is however not required. To cite the standard:

   > This identifier is intended to be machine readable and not necessarily meaningful to humans.

   Classification: The MessageID is *safe* and *blockable*. To block this protocol field, however, the message must be carefully parsed for `In-Reference-To:` headers and nested messages in every possible encapsulation. This makes blocking almost impossible in real–time, since a filter would have to dis–assemble the messages down to pure ASCII and then scan all these parts for indented, quoted or broken–up `MessageID` or `In-Reference-To` fields and substitute those. If the reference headers are to be meaningful, then those fields have to be re–substituted in answering messages. This could be accomplished without holding state on the filter, by simply encrypting the fields in the outgoing messages with some secret key and decrypting them on the way in.
   (b) **MIME multi-part delimiters** The Multipurpose Internet Mail Extensions define a `Content-Type: multipart/*` header [111], which tells the mail user agent to process this e–mail as a collection of messages. The different parts are separated by a delimiter which is defined *per e–mail* as a second parameter of the content–type header, called `boundary`. To cite the standard:

   > The boundary delimiter MUST NOT appear inside any of the encapsulated parts, on a line by itself or as the prefix of any line. This implies that it is crucial that the composing agent be able to choose and specify a

> unique boundary parameter value that does not contain the boundary parameter value of an enclosing multipart as a prefix. (*emphasis in the original*)

Classification: multi-part delimiters are *safe* and *blockable*. What applies for MessageIDs, however applies for delimiters: parsing through nested, quoted and broken–up messages makes blocking costly.

(c) **PGP Signatures with unknown key identifiers** Pretty Good Privacy (PGP) is the most popular e–mail encryption and authentication format with numerous implementations. The data format is specified in RFC 2440 [40]. One application of PGP is signing e–mails, by putting the signed text inside markers and computing a digital signature with a secret key over the framed text. The signature is appended to the text, so that the receiver can verify the signature. A PGP signature contains, among other fields, the *key id* of the public key to be used when checking the signature. The key id is a hash over the public key. If the key for the key id is not know by the receiver, he or she can query the network of *key servers*. It is however not unusual to find signatures with no publicly available key id. This allows to append a message to an e–mail, disguised as a PGP signature. The data fields are initialized as in signing a message, but the field holding the actual signature is filled with the hidden message, and the key id field is filled with random data.
Classification: Hiding messages in PGP signatures is *safe* and *blockable*. Without a public key, signature verification/falsification is impossible, and so is detection of the channel. Modifying PGP signatures by definition destroys the signature, which would defy the whole purpose of digital signatures. It is possible to enforce signature verification on every e–mail passing a server. This, however, leads to simple denial–of–service attacks where an attacker can force an SMTP server to perform expensive large–number operations.

2. **HTTP** Despite it's extreme popularity over the last years, HTTP's standardization went very slowly. The multitude of requirements that interested parties projected onto HTTP produced a complicated protocol with hundreds of headers and message types. The following features of HTTP Version 1.1 [113] are usable for covert communication:

   - **Server header** The `Server` header appeared in HTTP version 1.1. It allows the maintainer of an HTTP server to locate more than one HTTP *server name* behind a single IP address. The client indicates the name of the server in a special header. Since this field is ignored in most server setups, it can be used to embed data.
     Classification: The `Server` header is an *obvious* and *blockable*

channel. If an HTTP server replies to two requests with different `Server` headers with the same data, then it can be concluded that the header is unnecessary and can be filtered.

- **The CGI parameter part of a Unified Resource Locator (URL)** This is the location of the requested document, which is given in the `GET` statement in the request [114]. The URL may have a non-host-name, non-path appendix, which contains the parameters given to a Common Gateway Interface (CGI) script. This or the path section of the URL can be used as a covert channel. **Classification:** CGI parameters are a *safe* and *unblockable* channel.
- **Digest Authentication** Defined in RFC 2617 [97]. This is a challenge-response authentication sub–protocol. If a client requests a protected document from the server, it answers with a `401 Authorization Required` message, with an additional `WWW-Authenticate:` header, for example:

  ```
  HTTP/1.1 401 Unauthorized
  WWW-Authenticate: Digest
  realm="testrealm@host.com",
  qop="auth,auth-int",
  nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
  opaque="5ccc069c403ebaf9f0171e9517f40e41"
  ```

  This instructs the client to supply a username, the sent `nonce` and `opaque` values, and the MD5 hash of the password concatenated with the nonce. In the example (with password "Circle of Life"):

  ```
  Authorization: Digest username="Mufasa",
  realm="testrealm@host.com",
  nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
  uri="/dir/index.html", qop=auth,
  nc=00000001, cnonce="0a4f113b",
  response="6629fae49393a05397450978507c4ef1",
  opaque="5ccc069c403ebaf9f0171e9517f40e41"
  ```

  Classification The fields `nonce`, `opaque` and `response` are *safe* and *unblockable*.
- **Cookies** In its early versions, HTTP was a stateless protocol, to make implementation easier and failures on one side less likely to cause inconsistent state or memory leakage on the other. In an effort led by Netscape, Inc., a mechanism for keeping state on the client side was introduced in RFC 2109 [115]. To advise the client to keep a *(key,value)* pair as state for further communication, a server sends the `Set-Cookie` header in the reply to a request. This header may have optional fields, an example with all mandatory fields is:

  ```
  Set-cookie:  session-id-time=923385600;
  path=/; domain=.amazon.com;
  ```

```
expires=Tuesday, 06-Apr-2003 08:00:00 GMT
```

Classification: Cookies are *safe* and *blockable*. The abuse of cookies for user–tracing motivated the development of filtering proxies. Those programs are used to relay requests and replies between client and server, while they filter out certain headers. Filtering proxies such as `junkbuster` or `anonymizer.com` remove cookies and can be cascaded behind other HTTP proxies in a firewall. There are however a lot of services on the World Wide Web that become unusable if cookies are filtered.

- **Referer** This header is sent by the client and contains the URL of the document that links to the requested document. If the current document was generated by a CGI script and the script received its parameters by the `GET` method, then the CGI's path and parameters are part of the `Referer` in this document. It is also common practice to use the Referer as an additional parameter when designing CGI scripts. An example are so–called web–counters, where counters are kept (and returned as image) for every referring web page. It follows that the properties of the CGI parameters and HTTP cookies apply to Referer headers.
  Classification: The `Referer` header is a *safe* and *blockable* channel. As with cookies, removing the `Referer` from all requests will result in a reduced selection of working web–pages for the users behind the filter.
- **Other user–tracing techniques** A number of "Web Engineering" tools link the requests of a single client together, either by dynamically creating the URLs in the documents so that they contain a (website–) unique ID or by putting `HIDDEN` input fields in web forms, which are unique per displayed form. If those IDs were guessable, a client could take over the state of another, which would distort the click track. Probably for this reason, random–looking data can be encountered in the HTML code of CGI–generated web–pages.
  Classification: What has been said about URLs above applies again here. Embedding hidden fields in web forms or IDs in URLs is a *safe* and *unblockable* channel.

3. **DNS CNAMEs** The Domain Name Service maps host–names such as `ftp`.uni-erlangen.de to IP addresses (`131.188.3.71` in this case). DNS projects a system of *zones* on the more–or-less amorphous address space of IPv4. Hosts are grouped into zones (basically collections of domains) which are maintained on name servers. A special feature of DNS are `CNAME` resource records (RRs), in which nicknames for hosts can be stored. When a DNS server receives a request for the address of a nickname,

it replies with an answer containing the real host-name and its IP address. This feature has been used to implement covert channels. The outbound data is encoded in a host-name, e.g., `dv0q6xcif7mekidthpk0pxyy8x4.tunnel.info`. Inbound data is encoded into a reply, indicating that the requested host-name is a nickname, that its real name is some string containing the encoded data, and an IP address.
Classification: This channel is *obvious* and *unblockable*. Close scrutinity will show that a certain domain server (that of `tunnel.info` in the example above) has an uncommonly high amount of traffic. To block DNS requests, or to modify replies, would break protocols which include the host–name in application layer fields, e.g., HTTP version 1.1. There are some user–tracking systems which use a variation of the technique presented above to have each client use another hostname in the requests. They can then keep a table of the nicknames they gave out and data supplied by the clients to link client requests even in the presence of anonymizing proxies or dynamic IP addresses.

4. **DNSSEC SIG RRs** An extension of the DNS protocol provides a public key infrastructure for authentication of DNS information [85]. In this, resource records can be signed with a zone key, the signature is attached as a SIG record to the original record. If all zone's public key were known to the requesting parties, this should obviate most of the attacks on DNS. One of the signature algorithms defined in the DNSSEC RFC is DSA, so the subliminal channel shown in section 3.3.2 can be employed by the owner of a zone's secret key to subliminally communicate with requesting parties. Since the Time–to–Live for a particular record can be very short, this allows for some bandwidth. **Classification:** Hiding data in DNSSEC SIG records is *safe* and *unblockable*. Safeness is guaranteed by the hardness of the number–theoretic problem underlying DSA itself. To block this channel, it would be necessary to remove *all* DNSSEC extensions, thereby destroying the functionality of the protocol.

Marshalling and multiplexing protocols on top of the Application Layer

1. **XML** The extensible Markup Language [116] is a generalization of the Standard Generalized Markup Language [117]. Like the latter it is a meta–language to describe markup languages. There are eight working groups at the World Wide Web Consortium working on it; the 59 pages standard has 15 "Adjunct Specifications", and there are hundreds of drafts how to markup almost anything using XML. This plethora of standards makes it extremely difficult to filter documents with XML tags. What makes things even more interesting, is that the set of admissible tags and their attributes are defined outside the XML document itself. The definitions are referred to in the first section of every XML document. These external definitions are

most commonly referred through URLs, so some of the channels listed in section 2 can be used again.

Possible external documents to an XML document are:

(a) **Document Type Definitions**
The external definition of the tags and attributes in the document.
Example:
`<!DOCTYPE docelement SYSTEM"http://host.domain.com/path/type.dtd">`

(b) **XML Schemata**
A more convoluted version of the Document Type Definition. Schemas are written in XML themselves and therefore refer to their own DTD.
Example:
`<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">`

(c) **eXtensible Stylesheet Language Transformations**
Defines a set of transformations to be performed on the document and hints how to display the document.
Example:
`<?xml-stylesheet href="http://host.nu/path/style.xsl" type="text/xs"?>`

(d) **XML Name Spaces**
Define name spaces so that different parts of a document may contain identical tags without collisions or ambiguities. The referred document is not retrieved in standard application, however.
Example:
`<html xmlns="http://www.w3c.org/1999/xhtml>`

(e) **XSL Formatting Objects**
Describe in detail how elements inside the document should be rendered.
Example:
`<fo:rootxmlns:fo="http://definition.of/the/fo-language.used">`

(f) **XLINK**
Describes Hyperlinks between XML documents or portions thereof.
Example:
`<xlink simple xmlns:xlink="http://another.org/with/definition.s">`

To filter an XML document for dubious tags, attributes or attribute values, a filter would have to download all the external references. Some of the external definitions can in turn include external definitions, which makes this process even slower.

To show just one example of a covert channel in XML documents, we exploit an attribute type which can be found in many proposed markup languages. The attribute type `ID` describes a unique identifier; such an identifier can be as long as desired. The standard

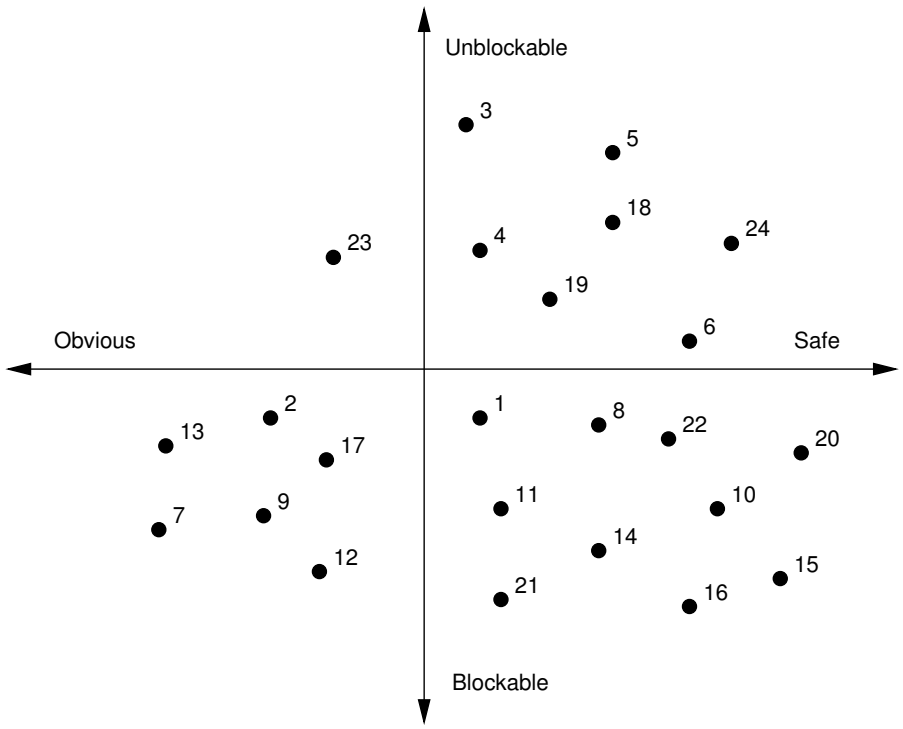

Figure 3.5: The presented channels as classified

defines a special data type for the values of `ID` attributes, stating that the values should start with a letter or an underscore (_) character. Apart from this, there are no restrictions on the values of the ID tag.

Since ID attributes can have corresponding IDREF attributes in other documents, a filter cannot simply discard or change IDs without affecting the usefulness of XML. The same argument applies as for the SMTP `MessId`, namely that a global, two-way filter is theoretically possible, but would be slow, hard to maintain and bound to fail for messages nested inside messages of a different type. It would also break authentication fields inside XML, such as XML signatures [118].

Another channel are the URLs for the DTDs or XML Schemata, which cannot be removed or substituted on the fly. Although it is possible to download every included file and to mirror those files on a local server under a different URL, that would again be slow and prone to errors.

Figure 3.5.2 shows the channels presented in this chapter together with their classification. The numbers correspond to the channels as follows:

1. IPv4 Identification
2. IPv4 Option

3. IPsec AH HMAC
4. IPsec ESP
5. IPv6 Source Address
6. IPv6 Flow Label
7. IPv6 Destination Options
8. IPv6 Fragment Identification
9. ICMP Echo Body
10. TCP Initial Sequence Number
11. TCP/UDP source ports
12. TCP Urgent Pointer
13. TCP Window Size
14. SMTP Message-ID
15. MIME Multi-Part Boundary
16. PGP Signatures
17. HTTP Server Header
18. CGI Parameter
19. HTTP Digest Authentication
20. HTTP Cookie
21. HTTP `Referer` Header
22. Entries in HTML Forms
23. DNS CNAME
24. DNSsec SIG Records

3.6 Conclusion

The list of covert channels presented in this chapter is by no means complete. All the channels are embedded in standard protocols and data formats. All those covertext protocols are in use now and have been implemented independently in various programs. Using these channels allows communication in presence of middle–boxes and strict content filters.

We conclude that

1. It is hard to design protocols which do not exhibit covert channels.
2. To re–design the existing protocols and data formats to close just some of the channels listed in this thesis might be possible. Forcing these changes on *all* hosts of the Internet, however, would mean a complete substitution of infrastructure, network stacks and application programs on all platforms. This is not to be expected.

If just some of TCP/IP's functionality is available, then there are covert channels. Many of these can escape the scrutinity of extremely powerful filters. It is therefore unreasonable to believe that filters can enforce policy on communication.

Chapter 4

Moving Points of Rendezvous

In this chapter we examine the problem of setting up a communication channel between two parties. Through the text we assume that the parties share a secret.

4.1 Introduction

The initiation of a channel between the parties intending to communicate is a critical point. If the IP addresses of the participating hosts are dynamically assigned or if some of the hosts are mobile, they have to use a service which we will call *point of rendezvous*, over which the hosts inform each other about their current addresses and perhaps other parameters before starting the communication. If there is no way to broadcast (or multicast) such information, all approaches have to use central services on machines with static addresses, one would suggest. This places the parties in a situation where their attempt to communicate may be overheard, blocked, or redirected so that they communicate through an eavesdroppers network (so–called man–in–the–middle attack).

We will show that the actual point of rendezvous accessed by the participants of the protocol need not be constant over time, central to the protocol or even predictable by anyone other than the participants, if the communicating parties share a secret. This is a reasonable assumption in many scenarios, since the wish to communicate electronically often follows a face–to–face meeting.

4.1.1 Common practice

The motivation for examining points of rendezvous is the same as for presence announcement protocols, with the additional restriction to avoid intermediate machines. To illustrate the approaches taken in current presence announcement protocols, we examine a few examples.

For example let us take the users of IP telephony (Voice over IP) from dial–up accounts and Internet cafes. They typically use presence announcement services such web–configurable dynamic DNS (see section 4.4.1 below)

or Netmeeting's[1] built–in not–quite–LDAP interface which in turn connects to central databases at Microsoft Inc. The respective partner in communication knows the unique ID of the user to be addressed in the scope of the naming service (ICQ's user id numbers, DNS's host-names, X.500 Distinguished Names in LDAP) and can thus query the announcement service for the transient address of their partner. The users need these alternative name spaces to attach the dynamically changing contacts to globally unique names.

The drawback is, of course, the involvement of intermediate services which are single points of failure and can easily be found and censored, or completely shut off.

4.1.2 Structure of this Chapter

In this chapter we present a general technique for announcing transient information such as dynamic IP addresses with existing client–server technology, while at the same time avoiding the inherent problems of single points of failure or censorship.

First we will list the requirements to be met, describe the general idea of our solution and then give a few examples how it can be adapted to current client–server protocols.

4.2 Points of Rendezvous

If examined closely, points of rendezvous do not need to store data for long periods of time, since the data in question is transient. It is also not necessary to store the data always in the same place as long as all interested and desired participants are able to find it. Globally scoped unique names over long times are also unnecessary, if it is possible to replace them with local identifiers or ones that change over time as long as the involved parties can compute the current identifier.

Minimum requirements for points of rendezvous are

1. Small amounts of data can be stored for short periods.
2. The data can be retrieved from anywhere for some time after it is written.
3. The key (e.g. Uniform Resource Locator (URL), address, identifier, ...) under which the data is stored can be computed by all desired participants.

4.2.1 Basic Approach

Our suggestion is to use existing free temporary storage on the Internet and in this wide range choose the actual place with a selector function, depending on the current time and other parameters.

[1] Netmeeting is a trademark of Microsoft, Inc.

Similar selector functions are used in steganographic methods such as Outguess [88, 119] to select the cover text bits suitable for embedding the hidden information. Our approach is a kind of reverse steganography: instead of hiding bits in the multitude of bits in a single cover text, openly storing data in the multitude of places on the net, so that it cannot be found either.

In the following sections we show how this works with different protocols as temporary storage. The protocols always follow the same pattern: first there is a setup, in which the parties agree on a shared secret and a selector function. With these, and clocks synchronized to a few minutes, one of the parties can call a write procedure. The selector function with the current time and the secret as parameter is used to find a set of location were data can be stored for a short time. The data to be communicated is written to this location. Corresponding to the write procedure, the other party calls a read procedure, which calls the selector function with time and secret as parameters to find the locations from which to retrieve the data, if there is any.

4.3 Example using Freenet

Freenet [120] is a good candidate for points of rendezvous. We described the protocol briefly in 2.3. Freenet was originally designed as a censure–proof distribution mechanism to further the right to free expression, it can be employed to further free communication, as we will show presently.

The following protocol implements a point of rendezvous on top of Freenet (it could be used for presence announcement as well):

Setup The parties agree on a shared secret S. Their clocks need to be synchronized to at least a few minutes.

Storage

1. The sending party A computes the hash $s_t = H(S, t)$ where t is the current time rounded to an appropriate position, e.g., an hour, and H is a cryptographically secure hash function, e.g., SHA1 [121].
2. A stores the message in Freenet under the Keyword Signed Key (KSK) (see 2.3) derived from the keyword s_t.

Retrieval

1. The receiving party B computes s_t as above.
2. B polls a nearby Freenet node for the document with the KSK derived from s_t.

This guarantees the following properties:

1. Only A and B can request the message, since the key under which it is stored was generated from the secret S.
2. A and B's name space for messages cannot be flooded with fake messages since they use random, equally distributed 160–bit values as names.

3. With S being secret and H a one–way hash function, only A and B can predict the key of the next message.

4. Freenet's caching algorithm will delete the message after some time, since it will be one of the most unpopular documents in the cache.

5. The nodes which store the message or pass it to B cannot read the message. This is an inherent property of Freenet.

If the idea is adapted to other protocols, the encryption has to be done by the participating parties.

The assumption of a shared secret is not unreasonable if private communication or cooperation is the intention. In such scenarios the participants are already acquainted with each other or can be introduced by acquaintances. The tremendously successful e–mail encryption protocol OpenPGP [40] relies on the observation that people normally write e–mails to persons they already know or who were recommended by friends.

4.4 Points of Rendezvous using established protocols

It is possible to implement points of rendezvous just with existing, well–supported client–server protocols, while at the same time avoiding single points of failure and censorship.

There are numerous services online that can be employed as temporary storage and indexed by an appropriately constructed selector function, although not all of these services are explicitly designed as such.

4.4.1 Short excursion: the Domain Name System

The domain name system (DNS) as described in RFC 1035 [112] is a kind of distributed database. The primary keys are *fully qualified domain names* (FQDN) as for example `ftp.uni-erlangen.de`. Names are resolved recursively from the right, which may need some explanation: There is a set of so–called *root servers*, which provide information on top-level domains (TLDs) as `.org`, `.int`, `.com`, `.mil`, `.gov`, `.edu`, `.net` and `.de`, `.uk`, `.va`, `.cx`, ... These are queried for the right–most part of the FQDN. From this information the search can be narrowed down, in our example to the server responsible for `uni-erlangen.de` and from there to the actual information about the host.

DNS defines several dozen of *resource records* (RRs) that can be associated with a name. The most common ones are

`A`	The IPv4 address of the host.
`MX`	The mail exchanger for a domain. This is a list of machines which are supposed to accept mail for the respective domain.
`NS`	The addresses of name servers responsible for that domain.
`SOA`	The *Start of Authority* record contains information about the administration of the domain.
`CNAME`	Indicates that the requested name is a nickname for another host or domain. The record contains the name of the host referred by the nick.

To avoid re–transmissions of the same data about popular hosts, DNS uses a hierarchical model and caching. A host normally does not request information from DNS itself but asks a local DNS server to do so. This server will first check for replies already seen for the request. Replies in DNS carry *Time–To–Live* (TTL) and *Expire* fields, entries older than their TTL are removed from the cache. If a server already has the requested information, it will send the cached reply instead of asking the respective servers.

The original system assumed that hosts are almost permanently connected. With the advent of dial–up connections and mobile hosts, this was no longer the case. In RFC 2136 [122] Dynamic DNS was defined for extremely short–lived entries. In these extensions, hosts register with the DNS servers of their domain to announce their current address and presence. The server in turn starts to reply to requests for the hostnames with adjusted records. Records thus created typically have a very short TTL.

4.4.2 Example using Dynamic DNS Services

Using always the same DNS name provided by a dynamic DNS service yields the possibility of censorship. The standard usage does also not allow the passing of arbitrary, however small pieces of data besides the standard DNS RRs. We can use multiple dynamic DNS providers to hide the messages and put small pieces of data in optional resource records.

The protocol instantiated for this temporary storage:

1. Setup: there are functions $F_0, F_1, \ldots F_{n-1}$ to automate account management at dynamic DNS providers $p_0, p_1, p_2, \ldots p_{n-1}$. The scripts can create, modify and delete entries for hosts with chosen names at those providers. Both parties have agreed on a shared secret S. Their clocks are sufficiently synchronized.

2. The sender A computes $s_t = H(S, t)$ where t is the current time rounded to an appropriate position.

3. A takes the lower 16 Bits of s_t, converts them to an unsigned integer j and computes $j \mod n - 1$ to get the index i of the provider to use.

4. A computes $p = H(s_t)$ and $h' = H(H(s_t))$ and base64–encodes [123] p and h' to derive a password and the host-name h.

5. A calls the function F_i to create an account for the host-name h at the dynamic DNS provider p_i.

6. If there is a short additional information to be passed, A puts it into the MX RR of h.

7. Cleanup: After the time interval indicated by the rounding in step 2, A calls F_i to delete the entries for host-name h.

For reading, B goes through steps 1 – 4 and then asks the DNS to resolve the host-name derived from h and the domain name served by p_i. For additional information B also requests the MX record for that host. Note that retrieving information by DNS provides receiver anonymity for hosts at the fringes of the net, because DNS requests are performed by the caching DNS servers of their ISPs and there is no practical incentive nor legal requirement to log DNS queries. Another advantage is that DNS resource records are removed automatically from the caches after their Time To Live [112] expires.

Short Excursion: Spam fighting

One of the features of the Simple Mail Transport Protocol (SMTP) — sending to multiple recipients — combined with the low price of bandwidth, gave rise to certain tele–marketing schemes. In these, a single sender uses the multiplying property of SMTP servers to flood the Internet with e–mails to *millions* of e–mail addresses without consent of the recipients. This way of free–riding is commonly called *Spam* [2]. Since there is a strong animosity against spammers in the Internet community, spammers try to hide their operations from their Internet Service Providers (ISPs). They often do this by exploiting *open mail relays*. These are mis–configured SMTP servers that accept and forward e–mails from anyone to anyone. The spam fighting community in turn has developed very sophisticated techniques for early detection and subsequent blocking of spam. These include centralized databases of currently known open relays, with the help of which filter rules for SMTP traffic can be dynamically constructed.

4.4.3 Using open relays for communication instead of spamming

If larger amounts of information are to be transfered, we suggest the following protocol:

Setup The initial setup is the same as above.

Sending

1. Sender A computes $s_t = H(S, t)$ with t being the time rounded to an appropriate position.
2. A gets a list of open mail relays from MAPS [125], ORBZ [126] or similar.

[2] The name goes back to a Monty Python sketch aired on 12/15/1970 [124].

3. A selects a number of open relays $r_0, \ldots, r_k$.
4. A creates a dynamic DNS host-name h as in the protocol above.
5. A sets the MX (Mail eXchange) record for hostname h to the address of a machine under A's control. This machine is configured to return a `308 Temporarily Unavailable` error on SMTP connections from the relays r_i.
6. A encrypts the additional information with s_t and sends it to `<name>@<`h`'s fully qualified domain>` once through each of the selected relays $r_0, \ldots, r_k$.
7. A deletes the entries for h at p_i after a period of time long enough for the r_j to deliver the mails.

Receiving

1. The receiver B goes through all the steps in the previous protocol to compute host–name and the password for that host at DNS provider p_i.
2. B starts a SMTP server on her machine, configured to receive e–mail addressed to host h.
3. B calls F_i to change the MX record of h to B's current IP address.
4. B waits for one of the relays to deliver A's message.
5. B decrypts the message.

Note that A's mail will remain in the spool directories of the relays until B resets the MX.

Both protocols are almost impossible to block. That is because A and B have dynamic IP addresses and dynamic DNS services are the last to block access from those. The sequence of host-names is unpredictable as long as S remains secret. The possible namespace is too big to be flooded(2^{160} possible names if SHA1[121] is used as hash function H). The communication pattern is such that an observer near either A of B cannot guess to who or from whom the respective host is sending or receiving messages.

4.4.4 Slight Degradation of Service

Another temporary storage available on the Internet in large quantities are guest–books on web pages. Research shows that most guest-books are implemented either by use of external servers or one of a few CGI scripts. This oligoculture gives rise to the following adaptation of points of rendezvous:

Setup There is a class of functions $F_0, F_1, \ldots F_{n-1}$ to automatically insert guest–book entries into guest–books and retrieve them accordingly for guest–books implemented by external servers [3] or scripts $p_0, \ldots, p_{n-1}$. Both parties share a secret S, the clocks are semi–synchronized as in the protocols above. A parameter r, the redundancy, is agreed upon.

[3]The URL of the actual guest-book is put in the `Referer` header to distinguish guest-books for different pages.

Sending

1. The sender A computes $s_t = H(S, t)$ where t is the current time rounded to an appropriate position.
2. A takes the lower 16 bits of s_t, converts them to an unsigned integer l and computes $l \mod n - 1$ to get the index i of the p_i (name of providing application server or CGI script) to use.
3. A sends an advanced search request to `altavista` or a similar search engine, asking for pages that have links to p_i. The returned list of URLs is stored in an array.
4. For all the URLs $u_0, \ldots, u_k$ in this array, A computes $h_j = H(s_t, u_j)$ and discards all u_j where the lower $\lfloor \log_2(k) \rfloor - r$ bits are not all zero. If this would discard all u_j, increase r by one and repeat.
5. For all the u_j still in the array A computes a heading $h_j = H(s_t, u_j)$. A truncates h_j to 64 bits.
6. The message to be transfered is encrypted in CBC mode with a suitable block–cipher with h_j as IV, and encoded with base64 encoding [123]. This gives a list of strings m_j.
7. A calls F_i to write the m_j to the guest-books u_j.

Receiving

1. Receiver B goes through the steps 1 – 5.
2. From all URLs in the list B polls the guest-books with function F_i and parses each of them for the base64–encoded truncated h_j. The entries starting with this IV are decrypted and the plain-text returned.

The selection in step 4 is done to assure that A and B use the same or at least an overlapping subset of the guest-books even if the search engine should reorder the list or add/remove some of the URLs.

Note how step 4 and step 5 make use of S and the hash function to guarantee that the subset of URLs remains secret and that all the written copies of the message are different apart from the length (which is a multiple of the block–cipher block length and for short messages will not really make the messages from different senders distinguishable).

The protocol places some burden on the maintainers of guest-books, and should therefore be used in practice sparingly.

4.4.5 ICMP Push and Pull

If one of the parties has a fixed address and the other is able to forge IP source addresses, it becomes possible to use ICMP as transmission channel. The Internet Control Message Protocol [105] states that the body of an "echo request" (icmp type 8) is to be returned in the reply. So if A know B's IP address, A can forge an ICMP echo request that appears to come from B. The body of the

forged datagram contains the message m to be transmitted. A sends this to an arbitrary host M which will copy the body into its reply and send it to B. To carry this one step further, if A and B have a shared secret s and their clocks synchronized to about 15 seconds, they can go through the following steps at a pre–arranged time t, rounded to the nearest quarter–minute:

1. A chooses a host M, the address of which is the output of a Pseudo-Random Number Generator seeded with t and s.
2. A pads the message m to a fixed length which is a multiple of 8 and appends a `\000` character.
3. A builds an ICMP datagram for M with B's address as source and the message as body. As stated in the RFC, the length of the datagram must be a multiple of 8 and thus the body is

$$\underbrace{m + \text{padding}}_{k\cdot 8\text{bytes}} + \underbrace{0 + \text{padding}}_{8\text{bytes}}$$

4. A fragments the datagram into two fragments of fixed length (following RFC 791 [127]) with an IP identification derived from the current time t and the secret s. The first fragment contains the message m and the first padding. The second consists just of the single character and seven bytes of padding.
5. A sends the first fragment to M (the *Push*) and discards the second.
6. B constructs the second fragment on his own. This is possible because B knows the number of the fragment (2), the length of the datagram, and can compute the address of M and the IP identification of the datagram.
7. B sends the second fragment to M (the *Pull*).
8. B receives the ICMP reply and extracts the message from the body.

At M, the first arriving fragment is stored in a buffer in the IP stack. If no second fragment with corresponding IP identification and source address should turn up in 30 seconds, the fragment is discarded. If a second fragment arrives, the two fragments are combined and the reassembled datagram is handed to the ICMP routines. These build a reply from the source address and the body, and send it to B. Since the two clocks are synchronized to at most 15 seconds, there is always enough overlap in time for reassembly.

Note that an observer, who taps routers near to A and observes just transport layer protocols (such as ICMP is), will never see A's datagrams, because they are incomplete and are discarded on the network layer. Since an observer cannot know M's address beforehand, communication cannot be blocked selectively. To forbid fragmentation of datagrams altogether would block a lot of legitimate traffic. For example, traffic that somewhere on its path enters VPN tunnels would be affected, because these always have a shorter Maximum Transfer Unit (MTU) than the standard 1500 bytes (which are characteristic of IEEE.802).

Reliability

In all of the suggested methods above, data is relayed through unwitting parties. It would be dangerous to rely on those if the data is of any importance. If multiple, say n, points of rendezvous are arranged between the parties, there are multiple algorithms to split data into n parts, k of which will suffice to reconstruct it, should $n - k$ of the relays prove unreliable, see for example [128, 129, 130].

4.5 Related methods from other fields

"Dead drops" for covert communication have been used by spies and/or criminals for hundreds of years. George Washington is reported to have used dead drops while directing the anti–British Culper spy ring, to name an example.

To build session keys from a long–time secret and a time–derived string has been used in the cryptographic community for a long time, see for example [131].

The "Warez" swapping community has used somewhat like moving points of rendezvous in recent years. Their approach in general is to first generate a lot of mirrors for the pirated software on hosting services such as Geocities and on poorly administrated FTP servers, and then to run scripts that check the availability and dynamically generate web pages with links to one of the mirrors that are not shut down yet. This web page contains the links to the pirated software and is itself hosted by a provider that is unresponsive to complaints, at least to those about links to "infringing material".

4.6 Results of this chapter

We have presented a general scheme for points of rendezvous that avoid dedicated central services and rely on unsuspicious, well–established protocols. From the scheme we produced several example protocols. The protocols do not require any new services and will traverse most firewalls and NATs. They are also very hard to filter even if their mode of operation is known.

Chapter 5

The Muted Post–horn

This chapter combines concepts from Chapter 2 and Chapter 3 to build a new protocol that creates an anonymous overlay network by exploiting the web browsing activities of regular users. We show that the overlay network provides an anonymity set greater than the set of senders and receivers in a realistic threat model. In particular, the protocol provides unobservability in our threat model. We also present a proof of concept implementation of the protocol which can be employed in current web–servers with minimal overhead.

5.1 Motivation

In chapter 2 we described several classes of protocols that provide anonymity or pseudonymity to senders and/or receivers. The currently most widespread anonymizing techniques for asynchronous communication are the "Cypherpunks" Type I and Type II e–mail anonymizers [36] which we described shortly in the same chapter.

Currently employed implementations of mix networks are, for example, the Cypherpunks [132] and Mixmaster [133] remailers and the nascent Mixminion project [41]. Chaum's work motivated other schemes which avoid expensive decryption at each step, to minimize delay, for example, Crowds [46] and Onion Routing [45].

In practice, most users of these systems do not run mix nodes themselves. They cause traffic patterns to and from the set of mix nodes, which a global, passive adversary can use to reduce the anonymity provided by the systems.

Although the amount of traffic inside the Type II mix network is quite impressive[1], the bulk of messages is generated by the nodes themselves. Educated guesses by the maintainers of six different remailers [2] [3] [4] indicate that about 80 percent of the traffic consists of "ping" messages to test the availability and delay of the nodes. On the positive side, this allows to hide user–

[1] the author's remailer, `passthru2`, carried peak traffic of more than 2000 messages per day.

[2] personal communication with the maintainers of `tonga`, `noisebox`, `rot26` and `nutshell` at the Privacy Enhancing Technologies Workshop in Dresden, March 27th, 2003.

[3] Personal communication with the maintainer of `squirrel` in January 2003.

[4] Author's experience while running the `passthru` and `passthru2` remailers 1998–2003.

generated messages in masses of automatically generated traffic inside the mix network. On the negative side, a global observer can ignore the inter–node traffic of the mix network and only trace the arrival and departure of messages from/to users. The anonymity set excluding the nodes themselves shows characteristic traffic patterns.

Possible attacks by a global, passive observer include intersection, timing and packet counting attacks on remailers and other systems derived from Chaumian mixes [34, 134, 35]. Suggested solutions introduce *cover traffic* into the protocols. This is achieved mainly by having the senders inject *dummy messages* [135], which are discarded at some mix.

In chapter 3 we presented a long list of covert channels in currently used protocols. One of the most interesting protocols studied there is HTTP. In contrast to other protocols, firewall administrators almost never block HTTP traffic initialized from the inside [5]. This is because the users of networked machines would then lose access to most of the Internet's repositories of information. This property in turn caused protocol designers to piggy–back their new protocols on HTTP, to avoid the problem of constant re–education of firewall administrators. An example is the Simple Object Access Protocol (SOAP) by Microsoft, Inc. The designers of SOAP chose to encapsulate their protocol inside HTTP to allow easy traversal of firewalls as explicitly stated in an early version of the specifications [137].

In this chapter, we present a Chaumian mix (see [30] and chapter 2) based entirely on HTTP protocol mechanisms, which we will call the *muted post–horn*[6] protocol. Its main application is building hidden community networks for delivering messages. Our design makes it hard to distinguish messages between the mix nodes from messages entering or leaving the mix network. It has the advantage that the anonymity set (see Section 2.1) consists not only of the participants in the mix protocol (nodes, senders, receivers), but also of a potentially large group of non-involved users of HTTP. These communicate with the mix nodes without being aware that they are transporting data. Another advantage results from this, i.e. that participants who do not run a node in the network can send and receive messages *unobservably*. In contrast to re–mailer networks, in our protocol, third parties transport the data between the mix nodes, so that there is never a direct connection between the nodes.

The rest of the chapter is organized as follows. In Section 5.3, we define our adversary model. Section 5.4 briefly explains the basics of Chaumian mixes again and explains why HTTP is a good choice of cover protocol. Related work is examined in Section 5.2. Section 5.5 describes a new class of covert channels inside HTTP which allow communication between servers under the cover of user–generated traffic. To show how these channels can be put to use in a Chaumian mix, we present a simple protocol in Section 5.6. In the following section, the protocol is refined against an active attack. Section 5.7

[5] "Server port 80 plagues Internet security" as Sam Costello put it in [136].

[6] in honor of a vaguely similar real–world network in Thomas Pynchon's novel "The Crying of Lot 49" [138].

discusses further enhancements that strengthen the protocol even against an unrestricted global observer. We conclude with Section 5.9.

5.2 Related Work

Using HTTP as substrate for other application level protocols is discussed in RFC 3205 [139], where only overt encapsulation of protocols is considered, naturally. There are several tools that tunnel protocols through HTTP, mostly to circumvent of firewalls, for example, Lars Brinkhoff's `httptunnel` [67]. These tools can be used to disguise any protocol as HTTP traffic, but the set of entities in which to hide — the *anonymity set* [28] — consists of just the sender and receiver, whereas the constructions listed in Section 5.5 use real cover traffic, involving unwitting web surfers as cover.

In Infranet [140], covert channels in HTTP are used to circumvent web–censorship. Web servers participating in the Infranet receive hidden requests for censored web pages and return the pages' content steganographically hidden in harmless images.

Goldberg and Wagner's TAZ and rewebber network [141] implements anonymous publishing based on HTTP. A simplified version of this is JANUS [142], where the requests are handled by a single node.

In Federrath et al.'s JAP [143], a mix network serves as web–proxy. The goal is to make web–surfing anonymous, not to store–and–forward messages with sender–receiver–unlinkability. In [143], the authors briefly touch on the subject of unobservability, but conclude that real users would inadvertedly destroy this property. Surveys such as Raymond's [35] mention the concept, but do not point to protocols that provide it.

5.3 Threat Model

To mount the timing and intersection attacks against many employed systems, the observer only needs to inspect the headers in the layers below the application layer of the TCP/IP stack at selected points on the Internet.

This precisely matches the capabilities of current (legal) telecommunication surveillance. To cite the CALEA[7] Implementation Section of the FBI:

> examine[ing] the full packet stream and examine protocol layers higher than layer 3 would place a high load on existing network elements in most architectures. [144].

The specification of "traffic data" in the EU Convention on Cybercrime [145] indicates that the intended mandatory surveillance by Internet service providers is restricted to the lower three layers of the TCP/IP stack.

We grant our adversary the additional ability to inspect application layer headers. This adversary model corresponds to an observer who is data–mining traffic logs for groups of communicating people.

[7] Communications Assistance for Law Enforcement Act

In Section 5.7 we show how our protocol can be hardened against a passive adversary who inspects the complete traffic.

Traffic Analysis

Traffic logs give away information about the social networks a person is part of. To extract this information, the data and a method for extraction has to be available to the adversary.

These traffic data logs are available in many countries, because they have to be kept by service providers in accordance to Article 17 and 20 of the Cybercrime Convention of 2001 [145]. Other countries have different legal frameworks but require traffic logs as well.

Early work on extracting data from traffic logs was done in military signal intelligence as early as 1940 as a reprint of a training manual of the U.S. SIGINT demonstrates [146]. More advanced techniques to extract meaning from e–mail traffic patterns were proposed in Schwartz and Wood [147] and later in Tyler et al. [148].

It follows that our adversary is a realistic and impending threat.

5.4 Background

In Chaumian mixes, nodes relay messages for each other. Each node has a (*public key*, *private key*) pair. To send a message along a chain of relaying mixes through the mix overlay network, the address of the final recipient is attached to the message. The result is encrypted with the public key of the last node in the chain. The address of the node is attached to the result and the process repeated for each node along the chosen path toward the first. On receipt of a message, a node decrypts it and — if it is not the final recipient itself — forwards it to the node specified in the decrypted text.

Later improvements on Chaum's scheme suggest random delays, various strategies to process and subsequently dispatch messages (*flushing*) [133, 37, 35], re–ordering of messages in the pool, padding the messages to a fixed size after decryption, and other improvements to ensure unlinkability.

Although recently contributed schemes (e.g. MorphMix [149], GNUnet's GAP [150] or Tarzan [151]) require users to transport traffic for other users, many deployed Chaumian mixes and derived systems suffer from the problem that most users do not, or cannot, run nodes in the systems themselves. They may be hindered by Network Address Translation [13], dynamic, and unstable, IP addresses or restrictive firewalling policies. This greatly weakens the achievable anonymity, as a passive adversary can observe traffic patterns leading to and coming from the mix network.

To thwart traffic analysis, we suggest hiding the protocol inside the well–established Hyper-Text Transfer Protocol (HTTP[113, 97]). According to recent measurements[152], HTTP accounts for the the second–highest percentage of data transfered on the Internet backbones, only slightly less than FastTrack's [153] file-sharing protocol.

Using HTTP as cover traffic brings another advantage. There is already an extensive body of research, and several implementations, which aim at providing some degree of anonymity for HTTP clients in the presence of various adversaries, see for example [46], [154], [155] and [143]. These techniques can be employed to enhance unlinkability.

HTTP is a client–server protocol. At first, this seems to imply that hidden data can only be forwarded through a chain of alternating clients and servers, all of which have to be participants of the hidden network. We will show, however, that communication between servers is feasible through standard web–clients which need not be part of the community using the covert protocol.

5.5 Server–to–Server Channel through unwitting Clients

Client machines as the hosts "at the fringes of the net" are generally considered to be end–points of data flows. This is reflected in the legal liability; for example, an ISP is not held liable for transit traffic as it is impossible or disproportionate to check every datagram, whereas users at their machines at home can be indicted based on evidence of traffic logs. In the following, we will prove as a corollary that those client machines do in fact carry transit traffic, and that users cannot reasonably be assumed to have control over it.

In this section, we explain how HTTP servers can communicate through clients without the consent or knowledge of the user. This constitutes a new class of covert channel, which transports data indirectly. The main mechanisms inside HTTP/HTML that allow such data transmissions are:

1. Redirects
2. Cookies
3. Referer[8] headers
4. HTML elements
5. "Active Content"

These features can be employed as follows:

Redirects

Redirects (RFC 2616 "303" messages [113]) are used to refer the client to another location. The location can be the URL of a CGI script, with optional parameters in the `QUERY_STRING` [156]. This allows CGI scripts to send data in said parameters to other CGI scripts through the browsers of unwitting web surfers. This channel's capacity is restricted to 1024 URL–encoded bytes [114].

[8] the typo was in the RFC and stuck.

Cookies

Cookies constitute a mechanism to keep state information on the client side. To advise the client to keep a `(key,value)` pair for further communication, a server sends a `Set-Cookie:` header in the reply to a request. The `value` part is allowed to be up to 4 kilobytes long, and the standard specifies that a client must be able to store up to a maximum of 40 cookies per server. In the server–to–server context, we can use optional features to transport data between the servers. The definition of cookies in RFC 2109 [115] defines a protocol sub–field `domain` which carries information about what group of web servers the cookie is to be sent to. The RFC states that the `domain` must contain at least two dots if it ends in a three–letter Top Level Domain (TLD) and at least three dots if it ends in a two–letter TLD. There are a many free Dynamic DNS services online, most of which provide hostnames in domains with this property, e.g., all hostnames in the zone administered by `dyndns.org` are in the same cookie domain. If a CGI script on server `foo.dyndns.org` sends a cookie of the form

```
KEY = VALUE; domain = .dyndns.org; Path = /;
```

to a browser and the browser connects to server `bar.dyndns.org`, then `bar` will get `foo`'s (key,value) pair. To get the browser to request data objects from `bar.dyndns.org`, the document requested from `foo` could contain one of the tags mentioned below under "HTML elements", or active content that requests data from `bar` automatically.

Referer

`Referer` headers tell the location of the web page or script that linked to the presently requested one. Since the naming of contents can be chosen arbitrarily by the server — and forced upon the browser by automatic requests as described below — this is another channel between servers through unwitting browsers. The length restriction of redirects applies here, too.

HTML Elements

The Hyper-Text Markup Language (HTML) version 4 includes elements that cause most browsers to automatically request given documents from HTTP servers. The following HTML tags and attributes have this property:

- `frame src=URL` Indicates a part of a `frameset`.
- `iframe src=URL` Defines an embedded frame.
- `img src=URL` Defines an inline image.
- `script src=URL` Indicates that JavaScript (see below) functions for this page should be loaded from `URL`.
- `link href=URL` Indicates out–of–band information for the current page.

- `object src=URL` Defines an embedded multi–media object to load.
- `applet codebase=URL` Indicates that Java (see below) classes for this page should be loaded from `URL`.
- `embed src=URL` Defines an embedded multi–media object to load.
- `layer src=URL` Defines a transparent layer of this page.

If the HTML document is created by a CGI script, the `URL` value in the tags above can be set to contain the address of another script together with parameters.
The `<META HTTP-EQUIV>` tag/attribute allows embedding of HTTP protocol header fields in the body of an HTTP message. This is useful for our purposes, because the header thus embedded in the body escapes the inspection of our adversary defined in Section 5.3. Interesting applications in our context are:

- Redirects (return code 303 [113]) inside successful replies (return code 500):
  ```
  <META HTTP-EQUIV="Refresh"
  CONTENT="3;URL=http://www.some.org/some.html">
  ```
  This line of HTML causes the browser to request `some.html` from `www.some.org` after 3 seconds.
- Setting cookies without a `Set-Cookie` header:
  ```
  <META HTTP-EQUIV="Set-Cookie"
  CONTENT="key=value;path=/;domain=.dyndns.org">
  ```
  This line sets a cookie on the browser, which will be transmitted to every server in the `dyndns.org` sub–domain to which the browser subsequently connects.

Active Content

So–called "Active Content" is code that is executed on the client. Currently used languages for active content are SUN's Java [157], Netscape's JavaScript [158], Macromedia's Flash [159] and Microsoft's ActiveX [160], the latter being restricted to a single browser, so it will not be discussed here. In Java's design, considerable effort was made to make the execution of untrusted code on the client secure. Java's security framework inhibits connections to servers differing from the one which supplied the running Java code , so it cannot be used to transmit data to different servers. Of the remaining two languages, we concentrate on JavaScript, because it is more wide–spread and better documented. Running code on unsuspecting surfers' machines opens a number of channels of varying bandwidth between scripts on servers. To name two examples:

- It is trivial to program redirects to CGI scripts (with parameters) in JavaScript.

```
<html>
<head>
<script language="JavaScript"><!--
function send_mesg () {
    var message = "@MESS@";
    document.subform.message.value=message;
    document.subform.submit();
}
//--></script>
</head>

<body>
<form name="subform" method="POST" action="@URL@">
<input type=hidden name="message" value="uninitialized">
</form>
<script language="JavaScript"><!--
send_mesg();
//--></script>
</body>
</html>
```

Figure 5.1: A JavaScript template that will cause a browser to send the message substituted for `@MESS@` to the URL substituted for `@URL@`.

- A script may construct an invisible `FORM` [161], fill the fields with data and send all of it to a CGI script in the body of a `POST` request without user interaction (see figure 5.1). This channel allows arbitrarily large payloads.

All the above mechanisms are heavily relied on by authors of HTML documents and CGI scripts.

5.6 The Muted Posthorn — A Chaumian Mix on Banner Adverts

To demonstrate how an anonymous messaging protocol can use HTTP as cover traffic to achieve unobservability against our adversary, we present a simple Chaumian mix.

5.6.1 The Setup

In our variant of Chaum's protocol, the *Muted Posthorn*, four (not necessarily disjoint) groups of entities are involved:

The node maintainers provide CGI scripts on HTTP servers. The scripts work as mix nodes and so every script has a $(publickey, secretkey)$ pair

and a pool for messages to be forwarded. A script is called with the message as the parameter of a POST request. The scripts work as in Chaum's mix networks, i.e. on receipt of a message, they decrypt it and look at headers specifying further processing. In our simple protocol, there are three possible actions, forwarding the message to another node, storing the message in a local mailbox with a supplied name (a 128 bit number), and sending the content of a given mailbox back to the requesting HTTP client. The outward visible action of the scripts is to return either an HTML document with JavaScript code that submits data to another node, or a short, static HTML document.

The linkers maintain web pages which all seem to contain the same small icon or banner advert. They do this by including an `iframe` which includes a `frameset` on one of the nodes. The `frameset` consists of a `frame` with the image and a second, invisible `frame`. This `frame` is created by a node and either contains the JavaScript code that does the actual transport, or the short HTML document.

The senders and receivers use this setup to communicate encrypted messages. Senders construct messages as in recent mix networks, e.g. Mixmaster [133], but the final delivery address of a message is always a mailbox on a node. A message thus constructed is sent to the first of the nodes in the chain by sending a POST request to a script. Receivers have to download their mailboxes' contents. They do this by sending encrypted "get mailbox number N" requests to the nodes where they keep mailboxes.

Hapless web surfers just visit the pages maintained by the linkers. Their browsers execute the JavaScript code returned by the node, transferring messages in the process.

5.6.2 A first Version

A simple variant of our protocol uses two kinds of messages:

`To:` messages contain encrypted messages to nodes in the network.

`Get:` messages request mailboxes from nodes.

Messages are always padded to a fixed length with randomness. When preparing a message m_0 for a sequence of nodes n_i, the sender recursively computes

$$m_{i+1} = \texttt{To}:||n_i||E_{n_i}(m_i).$$

where $E_n(m)$ encrypts message m for n's public key. For the last node, the `To` header is omitted. The sender submits the encrypted message to the last node in a POST request.

A receiving node tries to decrypt the message with its secret key. If decryption succeeds, the resulting text is parsed for headers.

If it is a `Get` message, the node looks up the requested mailbox. If it exists, the node throws a coin. On 0, the content is sent — through the requesting client — as a message to a random node in the mix network. The client can extract the message, for example, from its local browser cache. On 1, a fixed HTML response is sent to the client. If the mailbox does not exist, again a coin is tossed, this time to decide whether to send a randomly chosen message from the pool through the client or the HTML response.

If it is a `To` message, the address is examined. If it is a mailbox number, the message is stored in it. If the addressee is a URL, the message is put in the message pool for further delivery. Again, a coin throw decides whether a randomly chosen message from the pool or the fixed HTML response is returned to the client.

Against a passive observer as the adversary defined in Section 5.3, this protocol provides unobservability. Senders of messages and requesters of mailboxes send HTTP GET requests as any harmless client. Upon receipt of the JavaScript document, they substitute their own messages for the ones set inside the JavaScript code, and then let the browser execute the code. An observer who is restricted to the IP/TCP/HTTP headers thus cannot distinguish between harmless browsers and senders/receivers. This increases the anonymity set by the non-involved web surfers.

5.6.3 Improved Version

The simple protocol above is susceptible to a trivial denial of service attack. An adversary can simply request the frameset from a node repeatedly to drain its message pool.

To defend against this attack, we introduce acknowledgements (ACKs) for received messages between the nodes. Each message is kept in the pool and is re–sent until an ACK for the message is received. ACKs are not sent immediately, but are put in the message pool themselves.

An ACK should be tied to the message it acknowledges and to the node the message was addressed to, to avoid forged ACKs and replays. The standard approach would be to sign ACKs with the node's secret key. But deploying digital signatures at all would imply that the nodes know each others' public keys. Experience with remailers, however, shows that knowledge about such a global state of the mix network is hard to achieve and that it propagates slowly, if at all. For this reason we would like to avoid all public key operations at the nodes, except decryption.

Our suggestion is to send the hash of the decrypted text as ACK to the previous node (to make them indistinguishable from other messages, ACKs are padded with randomness to the fixed message size). The original sender knows all intermediate messages on the path, since she constructs them layer by layer. So she can inform every node on the path about what ACKs to expect. She does this by including the ACKs as values of additional `Ack` headers. The rule for constructing the next layer is now:

$$m_{i+1} = \texttt{To}:||n_i||\texttt{Ack}:||h(m_i)||E_{n_i}(m_i).$$

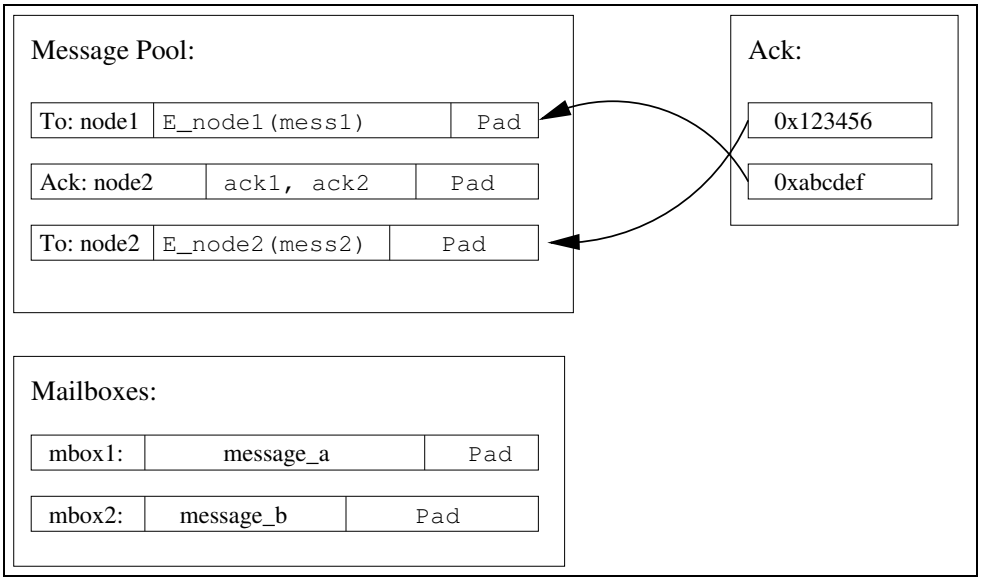

Figure 5.2: The internal state of a node: message pool with messages and acknowledgements for received messages, ACK table with outstanding ACKs and references to messages in the pool, and the mailboxes.

A node keeps three tables: the message pool of outgoing messages, a list of outstanding ACKs and a list of mailboxes (see figure 5.2).

On receipt of a message, a node checks if it is an acknowledgement. This is done by inspecting the first $|h()|$ bits of the message, where h is the cryptographic hash function used for ACKs. The resulting block is checked against the table of outstanding ACKs. If the block matches, the ACK itself and the message corresponding to it are removed from the table and the pool, respectively. Note that the URL of the sending node is transmitted by the client in the `Referer` header.

When processing `To` messages, the node now creates an entry in its ACK table with the value of the `Ack` header. The node computes the hash of the decrypted message and constructs an ACK message for the node that sent the message

5.6.4 Properties of the Protocol

The protocol inherits practical advantages from HTTP. All transactions of senders, receivers and unwitting web surfers can be performed through HTTP anonymizing systems such as Anonymizer [154], Crowds [46] or JAP [143].

The protocol's traffic is typically not blocked or modified at firewalls, and passes though Network Address Translation [13] without problems.

The coin tossing on the nodes makes the auto–submits terminate after two repetitions, in the mean. For a fixed message size of four kilobytes, the resulting traffic for the client is about the same as that for a banner advertisement (typically 16 kb).

5.7 Remaining Problems and Suggestions

Although the improved version seems promising, a number of open problems and possible enhancements must be addressed.

1. *Do acknowledgements (or lack thereof) introduce new points of attack?* If a node does not receive an ACK for a message, it will re-send the message at some later time. The repeating pattern marks it as being a message as opposed to an ACK or randomness. In our adversary model, this is no problem since the observer sees only the HTTP application headers. We considered several defenses against a stronger adversary. Techniques that transform the messages on the link between two nodes require either shared secrets between the nodes or a public key infrastructure (PKI). Experience with mixmaster remailers indicates that neither can be maintained in practice.

2. *Can we achieve unobservability against a global observer* who inspects complete data payloads, instead of just the headers? Yes, if the nodes refer to each other by `https` URLs. Hiding the compete application data from a passive adversary hides the repeating patterns mentioned in point 1 above, too. Note that we inherit the PKI from the secure HTTP protocol.

3. *User behavior influences the timing of message delivery.* This could lead to a Trickle [162] attack. If a client connects but the node's batching strategy does not dispatch a message from the pool, a node should send randomness of appropriate size to a randomly chosen node. This allows reuse of most of the known pooling algorithms (we implemented only the random pooling strategy).

4. *The time a message spends in the mix network before final delivery is dependent on external factors*, namely the whims and inclinations of unknown web surfers, and the willingness of web–site maintainers (linkers) to place links to the nodes on their pages. Should all the linkers' pages become unpopular at some point, communication would stop entirely.

   One way around this problem would be to combine the mix network with an Internet advertising company. The advertisements (placed in `IFRAME`s) would show ads while at the same time transporting data between the different servers of the advertising company. If cookies are used as the channel of communication, it would not be noticeably different from what Doubleclick Inc. is doing now [163].

5.8 The Implementation

We chose Perl [164] as the language of implementation. Perl has several advantages over other programming languages in the context of our work, the most significant being that most CGI scripts are implemented in Perl already. The support for Perl scripts in current web–servers is excellent [165], many

web–hosting businesses allow the installation of Perl scripts for their users. Perl has a highly modular structure and allows expansion of the language in run–time.

Encryption/decryption in our implementation is done envelope–style with authentication, the message is symmetrically encrypted with AES in CBC mode and a random session key. An HMAC of the message is appended, computed with a random authentication key. Session and authentication keys are encrypted with the public key of the recipient. The public key encryption uses RSA with random padding as defined in PKCS#1, version 2.0 [166].

The cryptographic algorithms we use were, however, not included in the standard modules of Perl installations. It turned out that the two available implementations of RSA–based public key cryptography for Perl, `Crypt::RSA` and `Crypt::OpenSSL::RSA`, were either hopelessly convoluted [9] or lacked essential features[10]. We choose to extend `Crypt::OpenSSL::RSA` by adding error reporting facilities and support for public key signatures. The modifications to this module were written in C and PerlXS [167] and were integrated into the main version shortly after [168].

To use the scripts that provide the functionality of the mix–nodes in our protocol, a user has to install the following non–standard Perl modules on the web server:

`Crypt::OpenSSL::RSA` This implements the RSA public key operations. It calls routines in the OpenSSL C–library [169], which is included in most operating systems used by web servers (FreeBSD, OpenBSD, Linux), and is available for almost every other system in use today.

`Crypt::CBC` This is a block–cipher–independent implementation of the Cipher Block Chaining mode. This module could well do with a complete rewrite, but since basic functionality is provided, we ignored this for the time being.

`Crypt::Rijndael` A fast C–based implementation of the AES block cipher.

`Digest::HMAC` A hash–function–independent implementation of a Hashed Message Authentication Code.

The code consists of the CGI script and a utility for sending and receiving messages through the system. Both use a small set of modules:

1. `Posthorn::Cryptofuncs` supplies the cryptographic algorithms.
2. `Posthorn::Message` is a class for parsing and processing messages.
3. `Posthorn::Spool` provides methods for storing, queuing and dispatching messages.

[9] `Crypt::RSA` depends on 13 (thirteen) other non–standard Perl modules and a large number theory package written in C, Henri Cohen's `Pari`.

[10] `Crypt::OpenSSL::RSA` version 0.11 did not have any error–reporting capabilities but simply died on errors.

5.9 Results of this chapter

Existing privacy enhancing technologies can assure anonymity only if the anonymity set is sufficiently large. In most current protocols, the size of the anonymity set is bounded by the number of the active users of the protocol. After defining a reasonable adversary model, we showed how the anonymity set of a protocol can be enlarged by having non–participants generate cover traffic. We presented new covert channels in the most wide–spread protocol on the Internet, the Hyper-Text Transfer Protocol, and proceeded to describe a simple Chaumian mix based on CGI scripts, in which the anonymity set consist of senders, receivers and unknowing participants, thereby enhancing anonymity for the senders and receivers.

Chapter 6

Conclusion

6.1 Is Filtering even possible?

In the last five years, there were repeated calls for surveillance and active filtering of Internet data traffic. These calls came from various parties, sometimes the media (mostly in the context of child pornography), politicians (in the context of their respective opponents), secret services (in the context of bigger budgets) and media conglomerates such as the Record Industry Association of America (RIAA) or the Business Software Alliance (BSA) (in the context of copyright violations).

The pleas to the legislative for filtering capabilities commonly take for granted that

1. data can be automatically recognized as belonging to a *semantically defined*[1] set of contents (e.g. child pornography, political hate speech, commands to terrorist groups, copyrighted material).

2. the originators cannot possibly bypass the filters.

3. the communicating parties can be somehow identified.

Whether or not the calls for mandatory filtering are appropriate or whether the presence of filters is desirable [2], is not the point here.

The main purpose of this work was to show that even in very restricted TCP/IP–based communications networks, free and anonymous communication is possible. In chapter 4 we showed how small setup–messages can be sent so that they are almost impossible to block, once that standard services of the Internet are made available. In chapter 3 we listed many possible covert channels within standard protocols of the TCP/IP stack, some of which are not detectable by per–datagram examination or cannot be blocked without destroying the functionality of the underlying protocol. In chapter 5 we showed how to implement a covert Mix–network solely on the all–pervasive HyperText Transfer Protocol, and that this Mix–network provides unobservability under normal circumstances.

[1] and extremely informal at that

[2] We totally acknowledge the necessity of spam–filters, for example, which again in practice demonstrate the fallacy of point 1.

Part II

Fair Validation of Database Replies

Chapter 7

Fair Validation of Database Replies

In this part of the Thesis, we present protocols for validating the replies of untrusted databases. Our main contributions are:

- A new cryptographic primitive, the *keyed hash tree*, which allows to summarize the state of a database including all the keywords *not* in the table.
- A protocol to validate a database reply against a signed state summary.
- Algorithms and data structures to store and update the necessary data at the database.

In Section 7.5 we compare our construction against related schemes.

In the following, the term *database* refers to a system supplying the simplest form of databases, a table of $(key, value)$ pairs with functions for keyed insertion and retrieval.

7.1 Motivation: Untrusted Databases

There are situations where the users of a database system cannot trust the database. This is the case if the database is outside the users' security perimeter, or does not follow the same security policies as the users. The users must therefore assume that the database could be taken over by an attacker.

The entities writing the database entries want to be sure that the database delivers their entries on requests unchanged. The database maintainer wants to be able to defend itself against accusations of misbehavior.

How can an attacker or malicious database maintainer deceptively influence the database's replies? The following list explains the attacks:

1. **Changing existing entries:** The attacker can change the value of an existing entry.
2. **Creating false entries:** The attacker can create $(key, value)$ pairs himself.

3. **Re–labeling existing entries:** The attacker can insert existing values under different keys.

4. **Returning old entries:** If the value of a $(key, value_1)$ pair is overwritten by a new $value_2$, the attacker can return the old $value_1$ on request for key.

5. **Denying existing entries:** On a *search* request, the attacker can make the database deny that a *value* exists for the given *key* in the table.

This is not purely of theoretical interest. Examples for such untrusted databases in real life are

Domain Name System The DNS [112] is a partially distributed database, where modification of entries can have devastating results. Attacks as described above are listed in [170]. This vulnerability of the critical component DNS led to the DNS Security Extensions [171].

Web Hosting Solutions There are different scenarios where a service provider hosts multiple customers' data on a central database. The entries are requested to construct a customer's web page on the fly on another machine. Even if the customer owns that machine, he or she must still trust the database to return all the entries unchanged.

Certificate Management If digital signatures bind to documents, then there is an incentive to modify the certificate authorities' (CA) revocation lists. The revocation list is a signed table of dates and revoked certificates. Modification allows a signer to declare his or her signatures void at a later point even though the key was not revoked at the time of signing. The signer could bribe the CA to back–date an entry.

7.1.1 Detecting Attacks

The readers should be able to detect if the database is lying about entries. The database should be able to prove the correctness of its answers as long as the writers are honest. For most of the listed attacks, this can easily be achieved if public key cryptography is employed. It is not necessary to assume the existence of a global Public Key Infrastructure for this purpose. It is sufficient if the readers of the database have access to all the public keys of the writers (presumably a smaller set than the readers). The keys must be stored outside the database. With this information at hand, and routines for signature creation (S_k) and verification (V_k), readers and writers can make attacks 1 – 3 detectable under standard cryptographic assumptions. Here is how:

1. **Changing existing entries:** Writers sign the values of their entries and include the signature in the *value* of the table. Readers check the signatures and detect changes by the database.

$$(key, value) = (key, \text{data}||S_k(\text{data}))$$

2. **Creating false entries:** The same applies here. Since the database has no access to the writer's secret keys, it cannot produce deceptive entries.

3. **Re–labeling existing entries:** This calls for an extension: the writers now include the *key* in the data they sign:

$$(key, value) = (key, \text{data}||S_k(key||\text{data}))$$

The readers can now check if the entry belonged to the *key* they gave in their *search* request.

Points 4 and 5, however, pose a harder problem. If there is no information about existing entries anywhere outside the database, then these attacks cannot be detected.

As for point 4, from the algebraic properties of standard digital signatures it follows that a signature on an old entry will still be valid after the entry has been replaced. There are signature schemes where the signer's cooperation is necessary for verification, called *undeniable signatures*. The concept was presented first by Chaum [172] in 1989. Such a scheme in our scenario would introduce much more communication than in the standard writer–database–reader setup. A writer who substitutes an existing entry would have to reliably notify the writer of the previous version, so that the previous author retracts the signature (e.g. by sending a new "signature outdated" message in the verification protocol). Readers would need to communicate with writers who might not be available every time someone wants to validate a reply. We would like to avoid such a complicated and error–prone setup.

The problem with point 5 is that from the perspective of the database, it would mean proving that it does *not know* the table entry, an impossible feat in this scenario. Since there is no *value* returned, there can be no signature, unless the writers supply a special "no entry" value tied to *every possible and unused* key.

There are ways around this, however, if we allow the writers and readers access to another service besides the database. The service is required to supply a single value on request; authenticated writers must be able to overwrite that value. We will call this service the *announcement service*;

The writers store a signed, condensed description of the database's state. This value must be updated after every write to the database. The readers can use this description to validate the database's replies. We will call this summary of database state a *state credential*. The validation protocol consists of two parts:

- A writer builds and publishes a new state credential from the previous one.
- Readers check the validity of a database reply. This requires interaction with the announcement service and the database.

7.2 State Credentials for Databases

Our requirements for the state credential (to be short and uniquely bound to the database state) suggest the application of a cryptographic hash function.

We will now describe several possibilities to build state credentials with hash functions.

7.2.1 Bloom Filters

Bloom Filters [173] provide means to verify set–membership *probabilistically*. The data structure used is a simple bit–vector of fixed length. To insert an entry into the filter, several indices into the bit–vector are computed by application of hash functions. The bits corresponding to the indices are set to 1. To check if a given object is in the set, the same functions are applied and the indexed bits checked. Should they all be 1, the object is considered to be a member. The state credential in our setting would be the bit–vector signed by the writer of the last entry. Depending on the size of the bit–vector, the output length of the hash functions and the number of entries, false positives are possible in Bloom Filters. This violates our requirement that the database must *always* be able to prove the correctness of its replies. Bloom filters were suggested for DNSsec in an IETF draft by Bellovin [174].

7.2.2 Simple Hash

This is the simplest form of a state credential. The writer queries the database for all entries and inserts the *key* of his/her new entry. The writer then sorts all keys in a pre–defined order and computes the hash value over all of them. This value is signed and supplied through the announcement service. Formally:

$$c = h(\sum_{i=0}^{n} k_i)$$

where n is the number of entries in the database, k_i is the ith *key* in the particular order and addition denotes string concatenation after adding a special "end of key" marker to the parameters. The simple hash is not a solution, because to check a database reply against this, a reader would have to download the whole database as well.

7.2.3 Hash Chain

Application of hash chains make the computation of the credential cheaper. The writer pulls the latest credential from the announcement service and "adds" the new entry's *key*. Formally:

$$c_0 = h()$$

$$c_i = h(c_{i-1} + k_i)$$

where c_i is the credential after adding the ith entry to the database and $h()$ is the hash of the empty string. Addition is defined as above.

To validate a database reply, a reader would have to pull all keys up to the requested key in their order of addition. In the interesting case of a non–existing key, the reader has to pull the whole database. So this is still not satisfactory.

7.2.4 Hash Trees

Merkle's [175] hash trees allow fast (Log-Time) verification of digests over many entries. Hashes over pairs of entries (the keys in our scenario) are computed and the resulting hashes are again paired and new hashes computed and so on. The number of entries has to be a power of 2. Formally:

$$H(l, j) = h(k_j)$$

$$H(i, j) = h(H(i+1, \lceil \frac{(i+j-1)}{2} \rceil).H(i+1, \lceil \frac{i+j+1}{2} \rceil))$$

where $l = \log_2(\#\text{entries})$, by "." we denote the concatenation of strings and k_j is the jth key in an arbitrary order ($H(1,1)$ is the root of the tree).

The state credential at the announcement service must be updated for each new entry and a hash tree root can only be computed if the number of entries is a power of two. So we must find a form of state credential that allows instant updating but still can use hash trees.

The solution is to split the number of entries into powers of 2 and to generate hash trees of appropriate height for all of those powers. See figure 7.2.4 for an illustration. For this reason, the state credential will consist of a number of hash tree roots, at most $\log_2(n-1)$ where n is the number of entries. When an entry is added to the non–empty database, and if the number of entries is odd after the addition, then a new hash tree of height zero (simply the hash of the key) is appended to the state credential. If the number is even, then the two most recent tree roots are combined (hashed together) to form the root of a higher tree. This is done recursively from the end. Note that we must start at the zeroth entry, since even an empty database needs a state credential.

So if the bit representation is

$$n-1 = \sum_{i=0}^{\lfloor \log_2(n-1) \rfloor} b_i \cdot 2^i$$

and $t = \sum b_i$ (the Hamming–weight of $n-1$), then the credential consists of the roots of t hash trees of height i for every $b_i = 1$.

The database in turn keeps a counter of entries. It can thus deduce in which of the trees in the current state credential an entry resides. This is achieved by comparing the entry's number against the current set of trees and their heights.

If a reader wishes to validate the databases' reply for a certain key k, the database has to supply a path in the tree leading to the leaf with k, as well as all pairs of hashes along that path. The reader can now verify that each pair's concatenation hashes to the corresponding hash in the next pair, up to the root of the tree in the state credential. Communication with the database for validation is bounded at worst by $\log_2(n) \cdot 2 \cdot |h()|$ bytes, where $|h()|$ is the byte–length of the hash's output.

The drawback of this scheme is that validation of a negative reply ("no such entry") still requires downloading the whole database.

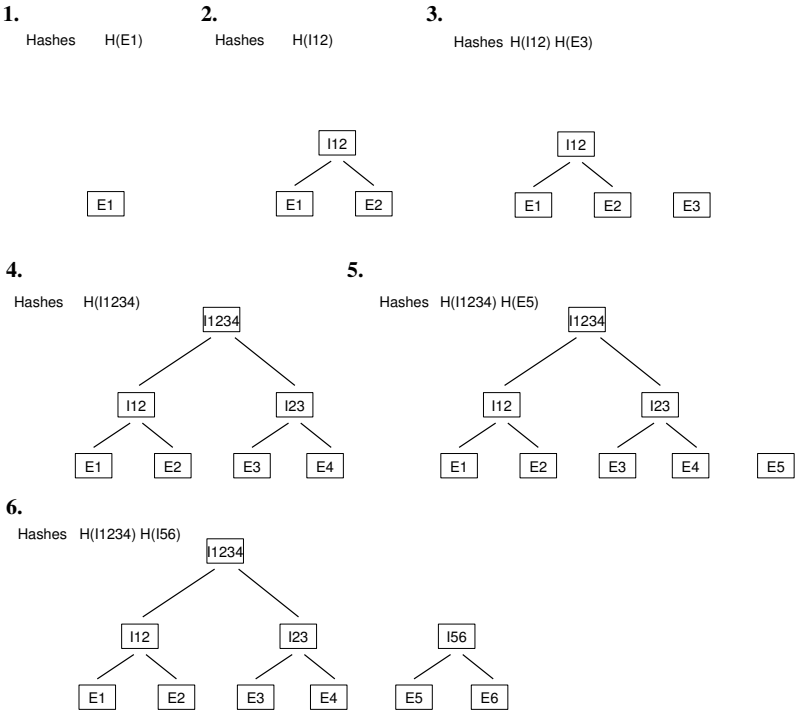

Figure 7.1: Hash tree attester for non–power–of–2 many entries

7.3 Keyed Hash Trees

Our contribution is the introduction of *keyed hash trees*. In these, the position of a leaf in the tree is dependent on the key corresponding to the leaf. Our construction uses a tree of height $|h()|$, where h is a cryptographic hash function. For every $(key, value)$ in the database, a leaf is inserted in the tree, where the path to the leaf from the root is defined by $h(key)$ and the data in the leaf is $h(value)$. Leaves without a corresponding entry are implicitly set to the hash of the empty string $h()$.

We call a sub–tree *empty* if there is no path leading through the sub–tree's root that leads to a leaf with a non–empty value. All leaves of an empty sub–tree have the value $h()$. The nodes in the layer above the leaves all have value $h(h(), h())$, and so on. Note that all nodes in a layer have the same value, which in turn is derived from the hash of the empty word. It is easy to pre–compute these values. It allows identification of a sub–tree as being empty by a single table lookup, since all empty sub–trees of equal height have the same pre–computable root hash.

From the collision resistance for cryptographic hash functions, it follows that

$$\forall a, b : a \neq b : h(a.b) \neq h(b.a).$$

The value of the tree's root thus depends on all the leaves' values and all the paths. In contrast to the schemes above, non–existing entries *do* have leaves and paths and can thus be validated.

Storing and working on a binary tree of height 160 (for $h = \text{SHA1}$ for example) seems daunting at first, since there would be $2^{160+1} - 1$ entries, only a few orders of magnitude less then the number of hydrogen atoms in the whole universe . But all except n of the leaves are empty, where n is the number of entries in the database. We will show in Section 7.4 that it is sufficient to store only the non–empty leaves and the branches leading to them from the root.

Before inserting a new entry $(key, value)$ in the database, a writer requests all pairs of hashes on the path to the new entry. The writer then validates the hashes against the current state credential, which is the root of the keyed hash tree. The writer substitutes the hash in the respective leaf by the hash of the *value*, and computes a new root hash. He then signs this root hash and puts it on the announcement service after inserting the new entry in the database.

To validate a database reply, the reader requests all pairs of hashes on the path derived from the *key*. For positive replies, the reader hashes over the returned value, compares that with the hash in the corresponding leaf and proceeds as with a standard hash tree.

If the reply was negative ("no such entry"), then the path must lead to an empty leaf and — as in the hash tree scheme above — the hash values can be checked recursively up to the root. The root must be the same as the signed value received from the announcement service.

7.4 Sparse Hash Tree Algorithms

Listings in pseudo–code for the routines described in this section can be found on pages 105 and following.

Our database should be able to respond quickly to requests for pairs of hashes along a given path. To do this, we need an internal representation of the keyed hash tree corresponding to the current database table.

We are helped by the keyed hash tree's property that all empty sub–trees of equal height are identical (an algorithm for creating a list of all empty sub–trees' roots is given in listing 7.1).

Instead of storing $2^k - 1$ nodes, where k is the length of the hash's output, we need to store only those nodes that are part of a path to an existing entry in the database, i.e. the hash of at least one key in the database describes a path leading through the node. To reduce the space further, we store only one node for every sub–tree that contains exactly one entry, i.e. the first node from the root down which lies on one single path is used to describe the whole sub–tree containing the corresponding leaf. We call the resulting data structure a *sparse hash tree* (see figure 7.4).

The main problem is that searching in the tree is done from the root down to the leaf, while the hash values in the nodes are computed from the leaves up.

Each stored node in our hash tree representation is a `struct` defined in listing 7.2.

The `b[2]` array contains pointers to the left and right children, or NULL if there is no respective child. `flag` indicates whether this node represents a

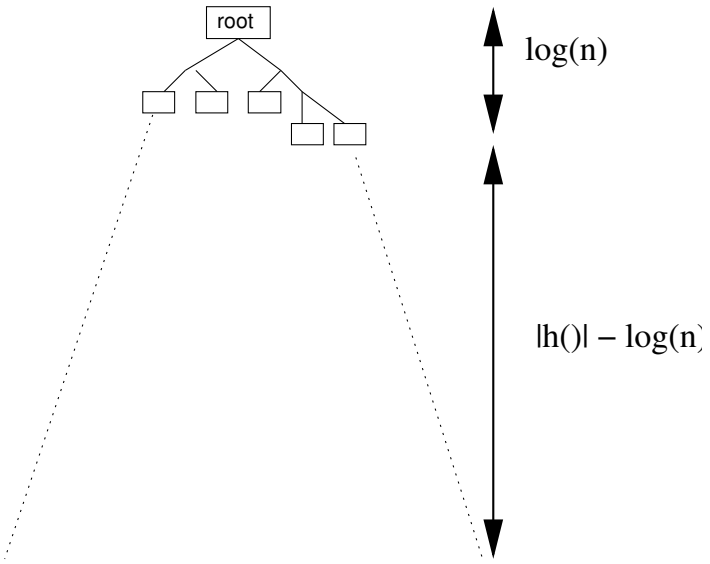

Figure 7.2: A sparse keyed hash tree reduced to the non–empty leaves.

`BRANCH` or a `LEAF`. `hash` is the hash of the tree rooted in this node. Only in leaf nodes is `e` not `NULL` and contains a pointer to a `struct entry` defined in listing 7.2. Note that leaf nodes do represent the leaves of the tree, but only non–empty leaves have leaf nodes. The nodes contain data structures that allow dynamic computation of all node–values of the hash tree below their `path`.

`entry` holds the information necessary to compute all nodes below the node pointing to it (see algorithm `leafnode` for details). `path` is a bit–string of `0`s and `1`s and describes the path to the actual leaf from the leaf node, i.e. the part of the path for which we don't store nodes, but generate them as needed. `value` contains the hash of the entry at the actual leaf.

7.4.1 Computing Pairs along a Path

The function `rootpath` defined in listing 7.3 on page 105 returns a list of pairs of hashes along a given path. The first two `If` statements handle the special cases that there is at most one entry in the sparse tree.

`rootpath` calls the auxiliary functions `nullnode` (listing 7.4), `leafnode` (listing 7.6) and `branchnode` (listing 7.7).

`nullnode` simply builds pairs of empty sub–tree roots in ascending order.

`leafnode` calls `singlepath` (listing 7.5) to generate a temporary list of all non–empty sub–trees below the leaf node. It then compares the supplied path against the path to the leaf and selects the pairs from the temporary list for the maximum left–match of the paths. For the rest of the path, empty sub–tree roots are appended to the selected pairs. This list is returned.

`branchnode` walks recursively down the tree as long as the path runs through BRANCH nodes. At each step it appends the hashes in the children of the current node to the list. At the point where the path changes to an un-

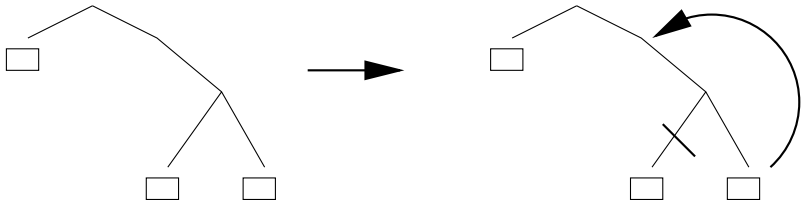

Figure 7.3: Deleting an entry and moving a neighbor to avoid dangling leafs.

defined or LEAF node, `branchnode` calls `nullnode` or `leafnode` respectively, to complete the list.

7.4.2 Inserting an Entry

`insert` (listing 7.8 on page 107) recursively walks down the tree along the given path, putting the traversed nodes on the stack. If the path runs into an empty subtree, a LEAF node with the given (path, value) is created. If the path runs through a LEAF node, the `entry` in the node is moved one step along its path down the tree, the node is converted to a BRANCH, and `insert` is called again on it. On the way up to the root, all the hashes on the path are adjusted.

7.4.3 Deleting an Entry

`delete` (algorithm 7.9 on page 108) recursively walks down the tree until it reaches the LEAF node with the entry to delete. On the way up, it checks if the current node has only one child, which is a LEAF. If so, the `entry` in the leaf is attached to the parent, and the child deleted. This is to handle situations where two leaves hang at the end of a stalk, as shown in figure 7.4.2. If one of the leaves is removed, the stalk would be a series of roots of subtrees with exactly one entry. To keep storage minimal, this should be avoided. `delete` attaches the lonely leaf at the uppermost branch of the stalk.

7.4.4 Properties

We assume that the hash function h behaves as an ideal hash function would, an assumption often used in cryptography called the Random Oracle Model. It implies that the output of h is indistinguishable from the output of a random function. Under this assumption, our data structures and algorithms have the following properties:

The sparse tree in memory is nearly balanced. If h's output behaves like randomness, the paths in the tree will be random walks starting at the root. For a large number n of entries, $\frac{1}{2^i}$ paths will lead through each node at the ith layer in the tree, and the average path length is $\log_2(n)$.

Space for the sparse tree is bounded by two times the number of entries in the database. We store n entries at n leaves, and each pair of nodes has one parent–node. This means that

$$\begin{aligned} maxnodes(n) &= \sum_{i=0}^{\lceil \log_2(n) \rceil} 2^i \\ &= 2^{\lceil \log_2(n) \rceil + 1} - 1 \\ &= 2 \cdot 2^{\lceil \log_2(n) \rceil} - 1 \\ &< 2 \cdot n \end{aligned}$$

Pairs along paths to a non–existing key can be computed quickly. If the key is not in the table, then the path will enter an empty subtree after $\log_2(n)$ steps, in the mean. For one million entries, for example, this means that after traversal of 20 nodes on average, `nullnode` will be called, and all that remains to be done is table lookups.

Readers recognize non–existing keys after $\log_2(n)$ steps from the root, in the mean. The pre–computation shown in 7.1 consists of k calls to the hash function, where k is the length of the hash's output. After this, a reader simply checks the returned pairs from the database against the pre–computed table `Empty`.

Maximum message size per validation is $2 \cdot |h()| \cdot |h()|$ bits. Two pairs of $|h()|$ bit hashes per step lie on a path with $|h()|$ steps. For SHA-1 as h, this would mean 6400 bytes per validation. If stronger collision resistance is required, a cryptographic hash function with longer output may be chosen. While the resistance should grow exponentially with the hash size, the messages grow only linearly.

The database can be distributed over several machines. The `insert`, `delete` and `rootpath` functions are independent of the actual height of the stored tree. As long as writers and readers know how to compute the $2^{d+1} - 1$ hashes at the root of the tree, they can use d independent databases. This would also allow d concurrent writes, by locking at the branches.

7.5 Related Work

There are other schemes with different objectives related to the one just presented. We will discuss the differences to our proposal.

7.5.1 Undeniable Attesters

Ahto Buldas et al. presented schemes for accountable certificate management in 2000 ([176] and [177]). In their scenario, the database is a part of a certificate authority (CA), i.e. the list of valid certificates. Their goal is to make sure that

there is never an ambiguity about the state of a certificate. On request for a key k, the database sends an *attester* $p = P(k, S)$, which is a cryptographic statement about the presence or absence of the key k in the table S. The database also returns a *digest* $d = D(S)$ which is a summary of the database's table S. Any party can then call a verification algorithm $V(k, d, p)$ which returns "Accept" if the statement p about k was correct for table summary d or else "Reject". Buldas et al. call an attester *undeniable* if a CA can produce two contradicting attesters for the same key with only negligible probability. Formally, this is defined as follows:

Let $\mathcal{EA}$ be the class of probabilistic algorithms of polynomial runtime. Let k be the security parameter (in our context the bit-length of the hash function's output). The attester $\mathcal{A}$ is given by the tuple of algorithms $\mathcal{A} = (G, P, D, V)$[1]. Its resilience against an attacker A of class $\mathcal{EA}$ is defined as

$$\mathrm{UN}_{\mathcal{A},k}(A) = \Pr[(x, d, p, \bar{p}) \leftarrow A(1^k) : V(x, d, p) = \text{Accept} \wedge V(x, d, \bar{p}) = \text{Reject}]\,.$$

If k can be chosen as to minimize $\mathrm{UN}_{\mathcal{A},k}$ below any given ϵ, then $\mathcal{A}$ is called undeniable.

The paper examines different schemes for attesters and concludes with an efficient, undeniable attester based on search trees.

Search Trees

A search tree (see for example Knuth [178]) is a binary tree with the additional property that there is a comparison relation $<$ and each node contains a value v for which the following holds:

$$v_l < v < v_r,$$

where v_l and v_r are the values in the left and right child of the node, respectively. If one or both children do not exist, the empty string is used instead. Buldas et al. add another field to each node, which holds the hash $h_v = h(h_{v_l}.v.h_{v_r})$. The digest d — the root of the search tree — is signed by the CA and published through an untrusted Publication Authority (PA). The digest corresponds to our state credential and the PA to our announcement service.

A proof p output by P for a key k is a list $p = (V_0, \ldots, V_m)$ of values $V_i = (h_L, k_i, h_R)$, where k_i is the value of a node and h_L, h_R are the hashes stored in the left and. right child of the node, respectively, or the empty string if there is no such node. The values are selected such that $k_0 = k$ if k is in the database. If k is not in the database, then the tree contains two keys j and l such that l is the smallest value larger than k and j the largest value smaller than k. By construction of the tree, there is a path from the node of j to the root leading through l or vice versa. The key of j and l which is lower in the tree is used as k_0 in that case. If k is larger or smaller than all keys in the search tree, then l or j is set to the largest or smallest key in the tree, respectively.

[1] G serves only to select a hash function for a given security parameter k.

p together with the published digest d allows to prove the absence of a key. The verifier can check the hashes from k_0 to the root and verify the strict order of child nodes. If a key k is not in the tree, then p will contain j, l such that $j < k < l \wedge \neg\exists k_i \in p : j < k_i < k \vee k < k_i < l$. By the strict order and lack of right children of j and left children of l, k cannot be a node's value in the tree.

The verifier $V(x, d, p)$ returns "Error" if any of the following fails:

- Checking the order of keys in p.
- Computing and comparing the hashes along the path given in p up to the root.
- Comparing d against the root hash in p.

If x is the first key in p, V returns "Accept", else "Reject".

Differences to our Proposal

The attester states absence or presence in a list, exclusively. A state credential includes the *value* of the entry under the *key*, so that changes of an entry can be expressed.

Buldas et al. assume that the database is reigned by a single entity which is trustworthy at the moment when an attester is issued, but may turn untrusted or unavailable later. That the database signs the digest itself establishes a different scenario than in our setting.

To show that the database cheated, a user has to find another attester contradicting the attester he/she received. In our setting, a reader can prove that the database cheated immediately and without contacting other readers.

Reduction of State Credentials to Attesters

We can build undeniable attesters for keys from signed state credentials. Since values of entries are irrelevant for attesters, we will substitute the $(key, value)$ pairs in our algorithms by (key, key) pairs. Since the empty word might be an entry's key in the generality of the proof, we will use the reserved key τ instead in that case. The digest d is the state credential, but is supplied by the database instead of the announcement service. Our (P, D, V) are defined as follows:

$P(x, S)$ is the list of pairs of hashes along the path given by $h(x)$ in the keyed hash tree generated from all entries in S.

$D(S)$ is the root of said keyed hash tree.

$V(x, d, p)$ outputs "Error" if any of the following fails:

- Comparing the hash at the leaf corresponding to x against $h(x)$ or $h()$.
- Hashing the pairs of hashes up along the path $h(x)$.
- Comparing the resulting root hash against d.

If the leaf value at the end of path $h(x)$ is the empty hash, then V returns "Reject". If the leaf value is $h(x)$, then "Accept".

Proof of Undeniability

Assume that some $A \in \mathcal{EA}$ outputs $(x, d, p, \bar{p})$ with probability ϵ, such that $V(x, d, p) =$ Accept and $V(x, d, \bar{p}) =$ Reject. Since V did not return "Error", all the pairs of hashes in p *and* in $\bar{p}$ resolved up to d. This means that A produced a collision in the hash h with probability ϵ, and the values $x_1, x_2, y_1, y_2 : (x_1.x_2) \neq (y_1.y_2) : h(x_1.x_2) = h(y_1.y_2)$ are members of p and $\bar{p}$. Thus we have reduced the security of our attester to the collision resistance of the hash h.

Considerations about the underlying Data Structures

Bulda's et al. use the search keys as they are, because the proofs of non–membership in the table rely on the greater–than relation between the keys. This has the disadvantage that the tree becomes unbalanced if the inserted entries are not uniformly distributed or if the first entry (the subsequent root) is not near the median of the set of entries. Unbalanced search trees cause longer searches, as more than $\log_2(n)$ nodes need to be traversed for some keys. The paper does not explain whether there is any re–balancing (see for example [178]) done on the search tree. Re–balancing this particular tree type would change all the hash–entries on the path from the previous root to the new one. Our sparse hash tree in memory is always balanced for large n.

Using the keys as they are makes the attester's size unpredictable. With state credentials, the size is fixed (and quite small).

7.5.2 Zero-Knowledge Sets

Unknown to this author, S. Micali, M. Rabin and J. Kilian presented a more general scheme, called Zero–Knowledge Sets [179] in 2003. Their goal was to prove set–membership non–interactively and that without leaking any other information about the set. The data structure underlying their scheme is almost identical to the keyed hash trees defined above.

Micali et al. use a commitment scheme to bind the database to its previous statements about its contents.

Pedersen's Commitment Scheme

In commitment schemes, a prover P *commits* to a message m by making public a commitment string c and computing a proof r. Before P publishes m or r, nothing should be inducible about the message m from c. After P sends message m, a verifier V can check whether this was the message committed to by c. For this, P supplies V with r. A commitment scheme is secure if P can produce contradicting proofs r, r' only with negligible probability.

Pedersen's scheme [180] requires four public parameters, i.e., two primes p, q such that $q|p-1$, and two generators g, h, where g and h generate $G \subset$

$\mathbb{F}_p^\star$, the subgroup of order q in $\mathbb{F}_p^\star$. P commits to m by publishing $c = g^m h^r \mod p$. r is the corresponding proof, a randomly chosen value $\leq q$. To verify the commitment, V re–computes c for herself and compares.

To produce two proofs r, r' for differing messages m, m', P would have to find r, r' such that $g^m h^r = g^{m'} h^{r'} \mod p$. Assume she succeeded. She then could compute $g^{m-m'} = h^{r'-r} \mod p$ and from this $(g^{m-m'})^{r'-r} = (h^{r'-r})^{r-r'} = h \mod p$ and thus get $log_g(h) \mod p$, breaking the discrete logarithm mod p. It follows that the security of the Pedersen commitment is reducible to the discrete log problem. It also follows that if $log_g(h)$ is known to the prover, she can fake arbitrary commitments.

Micali et al. use this scheme to construct a hash function $H : \{0,1\}^\star \rightarrow \{0,1\}^k$ where k is the bit-length of prime p. They use this hash function to process a data–structure very much like our keyed hash tree from section 7.3.

Their prover builds the tree by first inserting all leaves derived from the database's entries, such that the value in the leaf is $H(D(x))$, where $D(x)$ is the value stored in the database under key x, and the leaf's position in the tree is determined by the path described by $H(x)$. The prover adds all nodes on those paths to the root, their values remain undefined until later. She adds those nodes whose parents are now in the tree, but does not repeat this recursively. These nodes correspond to the empty subtree's roots in our construction.

In the next step, the prover generates a random exponent e_v for every node v in the structure, and stores $h_v = h^{e_v}$ in it if v is a non–empty leaf or the ancestor of one, and $h_v = g^{e_v}$ for the empty leaves. Thus the prover knows $log_g h_v$ for these nodes, a property exploited later.

The prover now computes commitments for the leaves by computing $c_v = g^{m_v} h_v^{r_v}$, where $m_v = H(D(x))$ or the empty string ϵ, h_v the value stored in the previous step, and r_v a random value chosen per leaf.

In the last step, the internal nodes of the tree get their commitment values generated. The process runs from the leaves upwards as in Merkle's scheme. Each internal node u is associated a value $m_u = H(c_{u0}, h_{u0}, c_{u1}, h_{u1})$, where $u0, u1$ are the left and right children of node u. From this and the value h_u stored already in u, c_u is computed as $g^{m_u} h^{r_u}$ for a randomly chosen $r_u \in \mathbb{F}_p^\star$. The commitment value for the whole database is the commitment of the tree's root c_D. This value is published before any queries are answered.

To prove that the value stored under key x is y in the database, the prover provides the tuples (m_v, e_v, h_v, r_v) for every node on the path through the tree given by $H(x)$, together with the values $c_{v0}, h_{v0}, c_{v1}, h_{v1}$ for v's siblings.

To check this proof, verifier V

1. compares the leaf's m_v against $H(y)$.
2. checks recursively that $m_v = H(c_{v0}, h_{v0}, c_{v1}, h_{v1})$
3. checks for every v on the path that $h_v = h^{e_v}$ and $c_v = g^{m_v} h_v^{r_v}$.

For a proof of the non–existence of key x in the database, the prover provides all the nodes' values along the path given by $H(x)$ as long as these nodes are in the generated tree. The last of these nodes, u, will have $m_u = 0$. If P

would supply this to V, then the verifier would learn that there are no non–empty nodes in this sub–tree. To disguise this fact, P generates a branch of non–empty nodes down to a virtual, empty leaf (the computation goes bottom–up however). Node u becomes the branch's root for this proof, the value $m_u = 0$ is substituted by the value in the uppermost node of the freshly generated branch.

The prover can do this convincingly because she can re–commit to any (m_u, r_u) in node u. This is caused by the special construction of h_u for empty leaves, where P knows the value of $log_g(h_u)$ by design. This allows to "glue" the generated branch of nodes, complete with their commitments, to node u.

The verification runs as above, except that V checks for $m_{H(x)} = 0$.

Differences to our proposal

In our proposal, the writers are different entities from the database, but they compute the commitment string. They are continuously updating and changing entries. In Micali et al.'s terminology, our writers *commit* to the history of their changes to the database, while the database later *proves* that it did reply in accordance to it state after the latest update. Zero–knowledge was no requirement in our design. Since the motivations and requirements are different, the mechanisms behind Micali et al.'s scheme and ours differ in several points:

Multiple, mutually trusting authors in our design, multiple database authors (writers) were a prerequisite. It is not obvious how zero knowledge sets could be adapted for multiple writers. Two writers would have to communicate at least the secret e_v, r_v for each node to be able to insert entries. This would require secure authenticated channels between all writers, which would raise the complexity tremendously.

Zero knowledge the zero–knowledge requirement is violated in our scheme, because a proof about any key will leak information about the existence or non–existence of other keys.

Database modification one of our requirements was that the database can be modified by the writers who can re–compute the credential/attester/-commitment for the database. The zero-knowledge property of Micali et al.'s scheme is lost if the database is allowed to change/add/remove entries. This is because the commitment values change along the path to a modified node and thus give away information about the minimum number of modified entries. This makes their construction extremely static and not suitable for our application. Micali et al. mention this in their open problems section.

Speed because Micali et al.'s scheme must satisfy additional constraints, it is much slower (five modular exponentiations per node versus one application of an optimized hash function). For a 1024 bit prime p, the computations per node are slower by a factor of 5780 [2]

[2]Measurement done on 32-bit Intel with the `libgmp` implementation of modular exponentiation [181] and the OpenSSL[169] SHA1.

7.6 Conclusion

Distributed databases such as DNS suffer from the problem that replies to queries can be modified maliciously by a database component that has been taken over. It follows that validation of replies against the original database writers' contents is necessary. To allow validation, digital signatures can be attached to entries; this obliterates most attacks. A remaining problem is the validation of "no–such–entry" replies, since there is no data to sign and no author responsible for it. In this part of the Thesis we presented an efficient validation mechanism that allows readers to verify the replies of a database, including those replies where the database denies the existence of the requested entry. To achieve this, we introduced a new cryptographic primitive, the Keyed Hash Tree, an extension of Merkle's hash trees. Our validation protocol is efficient, requiring only $2 \cdot |h()| \cdot |h()|$ bytes of validation data for an existing database entry and only $2 \cdot \log(n) \cdot |h()|$ bytes for non–entries, where $|h()|$ is the length of the hash's output and n the number of entries in the database. We described a space–efficient data–structure to store and process the keyed hash tree at the database together with algorithms for insertion, searching and deletion.

We examined the relationship to work in the area of certificate management, and showed how our proposal can be applied there as well. In comparison to the more general Zero–knowledge Sets by Micali et al., our scheme has the advantage that it is faster and allows subsequent modifications by multiple writers.

Listing 7.1: Pre–Computation of all empty sub–trees

```
list empty_init(void){
    i = 0;
    Empty[i] = h();
    for (;i<159;i++) {
        Empty[i] = h(Empty[i-1] . Empty[i-1]);
    }
    return Empty;
}
```

Listing 7.2: Structures `node` and `entry`

```
struct node {
    struct node *b[2];
#define LEAF   1
#define BRANCH 2
    int flag;
    struct entry *e;
    unsigned char* hash;
};

struct entry {
    unsigned char* path;
    unsigned char* value;
};
```

Listing 7.3: `rootpath()`: Generating all pairs of hashes along `path`

```
list rootpath(node root, bitstring path) {
    if(root == NULL){
        /* empty sub-tree */
        return nullnode(path);
    }
    If(root->flag == LEAF){
        /* exactly one leaf in subtree */
        return leafnode(root, path);
    }
    if(root->flag == BRANCH){
        /* non-unique path */
        init (List);
        return branchnode(path, root, List);
    }
}
```

Listing 7.4: `nullnode()`

```
list nullnode(bitstring path){
    init (List);
    for(i=0; i < length(path); i++){
        push List, (Empty[i], Empty[i]);
    }
    return(List);
}
```

Listing 7.5: `singlepath()`: Computing pairs of hashes in a sub–tree with exactly one given `(path, value)`

```
list singlepath(bitstring path, char value[160]) {
    revpath = reverse (path);
    init (List);
    init (pair);
    i=1;
    bit = shift revpath;
    pair[bit] = value;
    pair[not bit] = Empty[0];
    push (List, pair);
    while(bit = shift revpath) {
        pair[bit] = h(List[0][0] . List[0][1]);
        pair[not bit] = Empty[i++];
        push (List, pair);
    }
    return List;
}
```

Listing 7.6: `leafnode()`: Generating pairs of hashes along a `path` below a leaf `node`

```
list leafnode (struct node *node, bitstring path) {
    init (TmpList);
    init (List);
    init (pair);
    i=0;
    /* create list of non-empty nodes' hashes */
    TmpList = singlepath(node->e->path, node->e->value);

    /* compare given path against path of leaf */
    while(path[i] == node->e->path[i]) {
        append (List, TempList[i]);
        i++;
    }
    height = length(path) - i;

    /* if the two diverge, return empty nodes' hashes */
    while(height > 0) {
            pair = (Empty[height], Empty[height]);
            append (List, pair);
            height--;
    }
    return List;
}
```

Listing 7.7: `branchnode()`: Generating pairs of hashes along a `path` below a branch `node`

```
list branchnode (struct node *n, bitstring path) {
    init (pair);
    /* get hashes of children */
    for(b = 0, 1){
        if (n->b[b] != NULL){
            pair[b] = n->b[b]->hash;
        } else {
            pair[b] = Empty[length(path)];
        }
    }
    append (List, pair);
    bit = shift path;

    if (n->b[bit] != NULL && n->b[bit]->flag == BRANCH){
        /* walk down */
        append (List, branchnode(path, n->b[bit], List));
    } else {
        if(n->n[bit] != NULL && n->b[bit]->flag == LEAF){
            /* we hit a leaf */
            append (List, leafnode(path, n->b[bit]));
        } else {
            /* walked into an empty sub-tree */
            append (List, nullnode(path));
        }
    }
    return List;
}
```

Listing 7.8: `insert()`: Insertion of a new entry

```
struct node *insert(bitstring path, char value[160], struct node *node) {
    if (node == NULL ){
        /* virgin sub-tree: create new leaf */
        init (List);
        init (newnode);
        init (entry);
        newnode->flag = LEAF;
        List = leafnode(path, value);
        entry->path = path;
        entry->value = value;
        newnode->e = entry;
        newnode->hash = h(List[0][0] . List[0][1]);
        free(List);
        return(newnode);
    }
    if(node->flag == BRANCH){
        init (newnode);
        bit = shift path;
        /* walk down */
        newnode = insert(path, value, node->b[bit]);
        /* update the hash value on the way back */
        init (pair);
        pair[bit] = newnode->hash;
        if (node->b[not bit] == NULL) {
                pair[not bit] = Empty[length(path)];
        } else {
            pair[not bit] = node->b[not bit]->hash;
        }
        node->hash = h(pair[0] . pair[1]);
        return(node);
    }
    if(node->flag == LEAF){
        /* change LEAF to BRANCH */
        init (tmppath);
        init (tmpval);
        tmppath = node->e->path;
        tmpval = node->e->value;
        node->flag = BRANCH;
        /* move leaf's content down along its own path */
        free node->entry;
        node->entry = NULL;
        node = insert(tmppath, tmpval, node);
        /* insert new (path, value) in the now BRANCH */
        bit = shift path;
        init (newnode);
        newnode = insert(path, value, node->b[bit]);
        /* update hash on the way back */
        node->b[bit] = newnode;
        init (pair);
        pair[bit] = newnode->hash;
        if (node->b[not bit] == NULL) {
            pair[not bit] = Empty[length(path)];
        } else {
            pair[not bit] = node->b[not bit]->hash;
        }
        node->hash = h(pair[0] . pair[1]);
        return(node);
    }
}
```

Listing 7.9: `delete()`: Deleting the entry at the end of a given path

```
struct node *delete(bitstring path, struct node *node){
    if(node->flag == LEAF) {
        /* remove LEAF, stop recursion */
        free (node->entry);
        free (node);
        return NULL;
    }
    /* we're on the way to the leaf still */
    init (pair);
    /* remember the bit */
    bit = shift path;

    /* walk down */
    node->b[bit] = delete(path, node->b[bit]);

    /* reconstruct the path */
    path = prepend bit, path;

    /* how many non-children ?*/
    for (i = 0, 1) {
        if(node->b[i] == NULL) {
            emp = i;
            numemp++;
        }
    }
    if(numemp == 2){
        /* node was a stalk with a single leaf, delete */
        free (node);
        return NULL;
    }
    if(numemp == 1){
        if(node->b[not emp]->flag == LEAF){
            /* one child, a leaf, move it up */
            node->e = node->b[not emp]->e;
            node->e->path = node->e->path;
            /* correct the path */
            pop node->e->path;
            node->flag = LEAF;
            pair[emp] = Empty[length(path)];
            pair[not emp] = node->b[not emp]->hash;
            node->hash = h(pair[0] . pair[1]);
            free (node->b[not emp]->e);
            free (node->b[not emp]);
            return node;
        } else {
            /* branch node leading to at least two leafs */
            /* update the hash */
            pair[emp] = Empty[length(path)];
            pair[not emp] = node->b[not emp]->hash;
            node->hash = h(pair[0] . pair[1]);
            return node;
        }
    }
    if(numemp == 0){
        /* branch node with at least two leafs below */
        /* update the hash */
        pair[0] = node->b[0]->hash;
        pair[1] = node->b[1]->hash;
        node->hash = h(pair[0] . pair[1]);
        return node;
    }
}
```

Bibliography

[1] United Nations General Assembly. Universal declaration of human rights, 1948, 1998.

[2] European Convent. Draft for a europian constitution. `http://european-convention.eu.int/docs/Treaty/cv00850.en03.pdf`, Juli 2003.

[3] Parlamentarischer Rat. *Grundgesetz für die Bundesrepublik Deutschland*. May 1949.

[4] Reporters without Borders. 2003, a black year. `http://www.rsf.org/IMG/pdf/Round-up_RSF_2003.pdf`, January 2004.

[5] A. M. Odlyzko. Privacy, economics, and price discrimination on the internet. In N. Sadeh, editor, *ICEC2003: Fifth International Conference on Electronic Commerce*, pages 355–366. ACM Press, 2003.

[6] Hal R. Varian. Differential pricing and efficiency. *First Monday*, 1(2), 1996.

[7] Jamie Beckett. Scientology suit takes on cyberspace. *San Fransisco Chronicle*, March 1995.

[8] Electronic Frontier Foundation. EFF media release: EFF & scientists sue RIAA over censorship. `http://www.eff.org/Legal/Cases/Felten_v_RIAA/20010606_eff_felten_pr.htm%l`, June 2001.

[9] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[10] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless Inter-Domain Routing . RFC, Internet Engineering Task Force, September 1993. RFC 1519.

[11] W. Simpson. The Point-to-Point Protocol . RFC, Internet Engineering Task Force, December 1993. RFC 1548.

[12] R. Droms. Dynamic Host Configuration Protocol . RFC, Internet Engineering Task Force, March 1997. RFC 2131.

[13] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations . RFC, Internet Engineering Task Force, August 1999. RFC 2663.

[14] J. Postel. User Datagram Protocol . RFC, Internet Engineering Task Force, August 1980. RFC 768.

[15] Jonathan B. Postel. Transmission Control Protocol . RFC, Internet Engeneering Task Force, September 1981. RFC 793.

[16] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol . RFC, Internet Engeneering Task Force, October 2000. RFC 2960.

[17] R. Kent and R. Atkinson. Security Architecture for the Internet Protocol . RFC, Internet Engineering Task Force, November 1998. RFC 2401.

[18] Jerome Saltzer, David Reed, and David Clark. End-to-End arguments in system design. *ACM Transactions in Computer Systems*, pages 277–288, November 1984.

[19] B. Carpenter and R. Austein. Recent changes in the architectural principles of the internet. Technical report, Internet Architecture Board, February 2002.

[20] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification . RFC, Internet Engineering Task Force, December 1995. RFC 1883.

[21] Hal R. Varian. *Intermediate Microeconomics: A Modern Approach*. W. W. Norton & Co, 6 edition, 2003.

[22] E. Ostrom. Coping with tragedies of the common. *Annual Review Political Science*, 2:493–535, 2000.

[23] Jeff Chester. The death of the internet: How industry intends to kill the 'net as we know it. *TOMPAINE.com*, October 2002.

[24] David H. Crocker. Standard for ARPA Internet Text Messages . RFC, Internet Engeneering Task Force, August 1982. RFC 822.

[25] J. Myers and M. Rose. Post office protocol - version 3. `http://www.ietf.org/rfc/rfc1939.txt`, May 1996. RFC 1939.

[26] M. Crispin. Internet message access protocol - version 4rev1. `http://www.ietf.org/rfc/rfc3501.txt`, March 2003. RFC 3501.

[27] Mark R. Horton. UUCP Mail Interchange Format Standard. RFC, Internet Engineering Task Force, February 1986. RFC 976.

[28] Marit Köhntopp and Andreas Pfitzmann. Anonymity, unobservability, and pseudonymity — a proposal for terminology. In *Designing Privacy Enhancing Technologies*. Springer, July 2000.

[29] Rob Sherwood, Bobby Bhattacharjee, and Aravind Srinivasan. P5: A protocol for scalable anonymous communication. In *IEEE Symposium on Security and Privacy 2002*, pages 58–70, 2002.

[30] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24:84–88, February 1981.

[31] Adam Back. Hash Cash: A partial hash collision based postage scheme. `http://www.cypherspace.org/~adam/hashcash/`. Last Modified: Mon, 28 Oct 2002.

[32] HashCash. `https://ssl.dizum.com/hashcash/`.

[33] Matthew Wright, Micah Adler, Brian N. Levine, and Clay Shields. An analysis of the degradation of anonymous protocols. In *Proceedings of Symposium on Network and Distributed System Security (NDSS) 2002*, 2002.

[34] Lance Cottrell. Mixmaster and remailer attacks. `http://www.obscura.com/~loki/remailer/remailer-essay.html`, 1994.

[35] Jean-François Raymond. Traffic analysis: Protocols, attacks, design issues and open problems. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: Proceedings of International Workshop on Design Issues in Anonymity and Unobservability*, pages 10–29. Springer-Verlag, 2001. LNCS 2009.

[36] Lance Cottrell. Mixmaster. `http://www.obscura.com/~loki/`.

[37] Dogan Kesdogan, Jan Egner, and Roland Büschkes. Stop–and–go mixes providing probablistic security in an open system. In *Information Hiding: Second International Workshop*, pages 83–98. Springer, 1998. LNCS 1525.

[38] Claudia Diaz and Andrei Serjantov. Generalising mixes. In *Proceedings of the Third Workshop on Privacy Enhancing Technologies*. Springer, 2003.

[39] Steven Levy. *Crypto*. Viking, 2001.

[40] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP Message Format. RFC, Internet Engineering Task Force, November 1998. RFC 2440.

[41] George Danezis, Roger Dingledine, and Nicholas Mathewson. Mixminion: Design of a type iiianonymous remailer protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[42] Jonathan B. Postel. Simple Mail Transfer Protocol. RFC, Internet Engeneering Task Force, August 1982. RFC 821.

[43] T. Dierksa and C. Allen. The TLS Protocol Version 1.0. RFC, Internet Engineering Task Force, January 1999. RFC 2246.

[44] David M. Goldschlag, Michael G. Reed, , and Paul F. Syverson. Hiding routing information. In Ross Anderson, editor, *Information Hiding*, pages 137–150. Springer-Verlag, 1996. LLNCS 1174.

[45] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Onion routing for anonymous and private internet connections. *Communications of the ACM*, 42(2), February 1999.

[46] Michael K. Reiter and Aviel D. Rubin. Anonymous Web transactions with crowds. *Communications of the ACM*, 42(2):32–48, 1999.

[47] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030 – 1044, 1985.

[48] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[49] Manuel Blum. Coin Flipping over the Telephone. In *IEEE COMPCON 1982*, pages 133–137, 1982.

[50] Andrew S. Tanenbaum. *Computer Networks*. Prentice–Hall, 3 edition, 1996.

[51] Michael Waidner. Unconditional sender and recipient untraceability in spite of active attacks. In *Advances in Cryptography: EUROCrypt '89*, pages 302–319. Springer, 1989. LNCS 434.

[52] Jurjen Bos and Bert den Boer. Detection of disrupters in the DC protocol. In *Advances in Cryptography: EUROCrypt '89*, pages 320–327. Springer, 1989. LNCS 434.

[53] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptography: CRYPTO'2001*, pages 524–?? Springer, 2001. LNCS 2139.

[54] Adam Back. USENET eternity file system. `http://www.cypherspace.org/~adam/eternity/`, 1997.

[55] Ian Clarke. A distributed decentralised information storage and retrieval system. `http://freenet.sourceforge.net/index.php?page=theoppr`, 2000.

[56] The squid caching proxy. `http://www.squid-cache.org/`, 2004.

[57] A. Chankhunthod, P. B. Danzig, C. Neerdales, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. Technical report, Department of Computer Science, University of Colorado, Boulder, Colorado, USA, March 1994. Technical Report CU-CS-766-95.

[58] R. Hinden and S. Deering. IPv6 Multicast Address Assignments. RFC, Internet Engineering Task Force, July 1998. RFC 2375.

[59] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993.

[60] W. Fenner. Internet Group Management Protocol, Version 2. RFC, Internet Engineering Task Force, November 1997. RFC 2236.

[61] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC, Internet Engineering Task Force, July 1998. RFC 2373.

[62] David L. Mills. Network time protocol (version 3) specification, implementation and analysis. RFC, Internet Engeneering Task Force, March 1992. RFC 1305.

[63] David L. Mills. Simple network time protocol (sntp) version 4 for ipv4, ipv6 and osi. RFC, Internet Engeneering Task Force, October 1996. RFC 2030.

[64] Clay Shields. *Secure Hierarchical Multicast Routing And Multicast Internet Anonymity*. Dissertation in computer science, University of California, Santa Cruz, June 1998.

[65] International Standards Organization. Data communications – HDLC procedures – frame structure. ISO standard, International Standards Organization (ISO), 1979. ISO 3309.

[66] Sandra Steinbrecher and Stefan Köpsell. Modelling unlinkability. In *Proceedings of the Third Workshop on Privacy Enhancing Technologies*. Springer, 2003.

[67] Lars Brinkhoff. GNU httptunnel. `http://www.nocrew.org/software/httptunnel.html`.

[68] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[69] John McHugh. *Handbook for the Computer Security Certification of Trusted Systems*. Naval Research Laboratory, 1995.

[70] National Information Assurance Partnership (NIAP). Common criteria evaluation and validation scheme. `http://niap.nist.gov/cc-scheme/defining-ccevs.html`, 2004.

[71] Norman E. Proctor and Peter G. Neumann. Architectural implications of covert channels. In *Proceedings of the Fifteenth National Computer Security Conference*, pages 28–43, October 1992.

[72] R. A. Kemmerer. Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels. In *ACM Transactions on Computer Systems*, volume 1:3, pages 256–277, August 1983.

[73] Ira S. Moskowitz and Myong H. Kang. Covert channels - here to stay? In *Compass'94: 9th Annual Conference on Computer Assurance*, pages 235–244, Gaithersburg, MD, 1994. National Institute of Standards and Technology.

[74] Gustavus J. Simmons. Verification of treaty compliance revisited. In *Proceedings of the 1983 IEEE Symposium on Security and Privacy*, pages 61–66. IEEE Computer Society Press, April 1983.

[75] Gustavus J. Simmons. The subliminal channel and digital signature. In *Advances in Cryptology, Eurocrypt 1984*, 1984. Lecture Notes in Compuer Science 209.

[76] G. Simmons. Subliminal channels: Past and present. *European Transaction on Telecommunications*, 5(4):459–473, 1994.

[77] U.S. Department of Commerce. Digital signature standard (DSS). Technical report, National Institute of Standards and Technology, May 1994. FIPS Pub 186.

[78] Uriel Feige, Amos Fiat, and Adi Shamir. Zero knowledge proofs of identity. In *Proceedings 19th ACM Symposium on the Theory of Computing*, pages 210 – 217. ACM Press, May 1987.

[79] Ross Anderson. *Security Engineering*. John Wiley & Sons, 2001.

[80] Ross Anderson, Serge Vaudenay, Bart Preneel, and Kaisa Nyberg. The newton channel. In *IWIH: International Workshop on Information Hiding*, pages 143–148. Springer, 1996. LNCS 1174.

[81] M. Burmester, Y. G. Desmedt, T. Itoh, K. Sakurai, H. Shizuya, and M. Yung. A progress report on subliminal-free channels. In Ross Anderson, editor, *Information hiding: first international workshop*, pages 157–168. Springer, 1996. Lecture Notes in Computer Science 1174.

[82] Yvo Desmedt. Simmons' protocol is not free of subliminal channels. In *9th IEEE Computer Security Foundations Workshop*, pages 170–175, June 1996.

[83] R. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[84] T. ElGamal. A public key cryptosystem and signature scheme based on discretelogarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[85] D. Eastlake and C. Kaufman. Domain Name System Security Extensions. Technical report, Internet Engeneering Task Force, August 1982. RFC 2065.

[86] Peter Gutmann. X.509 style guide. `http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt`, October 2000.

[87] National Institute of Standards and Technology (NIST). Secure hash standard. Technical report, National Institute of Standards and Technology (NIST), April 1995. Federal Information Processing Standards 180-1.

[88] Niels Provos. OutGuess - Universal Steganography. `http://www.outguess.org/`, August 1998.

[89] CompuServe Incorporated. GIF-89a specification. `http://www.w3.org/Graphics/GIF/spec-gif89a.txt`, July 1990.

[90] Fabien A.P. Petitcolas, Ross J. Anderson, and Markus G. Kuhn. Attacks on copyright marking systems. In David Aucsmith, editor, *Second International Workshop on Information Hiding*, pages 219–239. Springer, 1998. LNCS 1525.

[91] Ronald L. Rivest. Chaffing and winnowing: Confidentiality without encryption. *CryptoBytes*, 4(1):2–17, 1998.

[92] Ross Anderson. *The Bell–LaPadula Security Policy Model*, chapter 7.3, pages 139–146. In Security Engineering [79], 2001.

[93] National Information Assurance Partnership (NIAP) Interpretations Board. Covert channel capacity limit policy. `http://niap.nist.gov/cc-scheme/PUBLIC/0113.html`. Public Interpretations Database, entry I-0113.

[94] Microsoft Corporation. Rich text format (RTF) specification, version 1.6. `http://msdn.microsoft.com/library/en-us/dnrtfspec/html/rtfspec.asp`, May 1999.

[95] Jeffrey S. Havrilla, Cory F. Cohen, Roman Danyliw, and Art Manion. Statistical weaknesses in TCP/IP initial sequence numbers. `http://www.cert.org/advisories/CA-2001-09.html`, September 2001. CERT Advisory CA-2001-09.

[96] J. Myers. Simple Authentication and Security Layer (SASL). RFC, Internet Engineering Task Force, October 1997. RFC 959.

[97] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication . RFC, Internet Engeneering Task Force, June 1999. RFC 2617.

[98] Paul J. Leach and Dilip C. Naik. A common internet file system (CIFS/1.0) protocol (preliminary draft). Technical report, Internet Engineering Task Force, March 1997. IETF Draft.

[99] Daniel Hartmeier. Design and performance of the openbsd stateful packet filter. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference (FREENIX '02)*, 2002.

[100] S. Kent. Security Options for the Internet Protocol. RFC, Internet Engineering Task Force, November 1991. RFC 1108.

[101] Ross Anderson. *Multilevel Security*, chapter 7, pages 137–160. In Security Engineering [79], 2001.

[102] S. Kent and R. Atkinson. IP Authentication Header. RFC, Internet Engeneering Task Force, November 1998. RFC 2402.

[103] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification . RFC, Internet Engeneering Task Force, December 1998. RFC 2460.

[104] T. Narten and R. Draves. Privacy extensions for stateless address autoconfiguration in ipv6. `http://www.ietf.org/rfc/rfc3041.txt`, January 2001. RFC 3041.

[105] Jonathan B. Postel. Internet Control Message Protocol. RFC, Internet Engeneering Task Force, September 1981. RFC 792.

[106] J. Christian Smith. Covert shells. `http://rr.sans.org/covertchannels/covert_shells.php`, November 2000.

[107] L. Joncheray. Simple active attack against TCP. In *Proceedings of the 5th USENIX UNIX Security Symposium*, June 1995.

[108] Theo de Raadt, Niklas Hallqvist, Artur Grabowski, Angelos D. Keromytis, and Niels Provos. Cryptography in OpenBSD: An overview. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.

[109] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. *ACM Computer Communications Review*, 29(5):71–78, October 1999.

[110] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. `http://www.ietf.org/rfc/rfc1323.txt`, May 1992. RFC 3041.

[111] N. Freed and N. Borenstein. Multipurpose internet mail extensions (MIME) part two: Media types. RFC, Internet Engeneering Task Force, November 1996. RFC 2046.

[112] P. Mockapetris. Domain Names–Implementation and Specification. RFC, Internet Engineering Task Force, November 1987. RFC 1035.

[113] UC Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt`, June 1999. RFC 2616.

[114] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). RFC, Internet Engineering Task Force, December 1994. RFC 1738.

[115] D. Kristol and L. Montulli. HTTP state management mechanism. `http://www.ietf.org/rfc/rfc2109.txt`, February 1997. RFC 2109.

[116] XML Core Working Group. Extensible Markup Language (XML). Technical report, World Wide Web Consortium, 2002.

[117] Subcommittee JTC 1/SC 34 International Organization for Standardization. Standard generalized markup language (SGML). Technical report, International Organization for Standardization, 1986. ISO 8879:1986.

[118] Donald Eastlake 3rd, J. Reagle, and D. Solo. (Extensible Markup Language) XML-Signature Syntax and Processing. RFC, Internet Engineering Task Force, March 2002. RFC 3275.

[119] Niels Provos. Probabilistic methods for improving information hiding. Technical report, University of Michigan, Center for Information Technology Integration, January 2001. CITI Technical Report 01-4.

[120] Ian Clarke. Freenet. In *Peer-to-Peer, Harnessing the Power of Disruptive Technologies*, pages 123–132. O'Reilly & Associates, March 2001.

[121] D. Eastlake the 3rd and P. Jones. US secure hash algorithm 1 (SHA1). `http://www.ietf.org/rfc/rfc3174.txt`, September 2001. RFC 3174.

[122] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. Dynamic updates in the domain name system (DNS UPDATE). RFC, Internet Engeneering Task Force, April 1997. RFC 2136.

[123] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One . Rfc, Internet Engeneering Task Force, November 1996. RFC 2045.

[124] Graham Chapman, John Cleese, Terry Gilliam, Eric Idle, Terry Jones, and Michael Palin. *Viking Sketch*, volume 2, page 27. Methuen, 1989.

[125] Paul Vixie. Maps relay spam stopper. `http://work-rss.mail-abuse.org/rss/index.html`.

[126] Ian Gulliver. Open relay blackhole zones. `http://www.orbz.org/`.

[127] Jon Postel. Internet protocol. RFC, Internet Engineering Task Force, September 1981. RFC 791.

[128] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrival. In *Workshop on Distributed Algorithms (WDAG '97)*, pages 275–290. Springer, September 1997.

[129] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2), April 1989.

[130] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, November 1979.

[131] Bruce Schneier. *Applied Cryptography (Second Edition)*, volume 1. John Wiley & Sons, 1996.

[132] S. Parekh. Prospects for remailers. *First Monday*, 1, August 1996.

[133] Ulf Möller and Lance Cottrell. Mixmaster Protocol — Version 2. Unfinished draft, January 2000. http://www.eskimo.com/~rowdenw/crypt/Mix/draft-moeller-mixmaster2-proto%col-00.txt.

[134] Adam Back, Ulf Möller, and Anton Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In Ira S. Moskowitz, editor, *Information Hiding (IH 2001)*, pages 245–257. Springer-Verlag, LNCS 2137, 2001.
http://www.cypherspace.org/adam/pubs/traffic.pdf.

[135] O. Berthold and H. Langos. Dummy traffic against long term intersection attacks. In R. Dingledine and P. Syverson, editors, *Privacy Enhancing Technologies (PET 2002)*. Springer, 2002. LNCS 2482.

[136] Sam Costello. Server port 80 plagues internet security. *InfoWorld*, April 2002.

[137] Aaron Skonnard. SOAP: The simple object access protocol – MIND january 2000. http://www.microsoft.com/mind/0100/soap/soap.asp, January 2000.

[138] Thomas Pynchon. *The crying of lot 49*. J.B. Lippincott Company, 1966.

[139] K. Moore. On the use of HTTP as a substrate. Technical report, Internet Engeneering Task Force, February 2002. RFC 3205.

[140] Nick Feamster, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan, and David Karger. Infranet: Circumventing Censorship and Surveillance. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[141] Ian Goldberg and David Wagner. TAZ servers and the rewebber network: Enabling anonymous publishing on the world wide web. *First Monday*, 3(4), August 1998.

[142] Andreas Rieke and Thomas Demuth. JANUS: Server anonymity in the world wide web. http://www.thomas-demuth.de/veroeffentlichungen/eicar2001.pdf, March 2001.

[143] Oliver Berthold, Hannes Federrath, and Stefan Köpsell. Web MIXes: A system for anonymous and unobservable internet access. In Hannes Federrath (Ed.), editor, *Designing Privacy Enhancing Technologies. Proc. Workshop on Design Issues in Anonymity and Unobservability*, pages 115–129. Springer, 2001. LNCS 2009.

[144] Commitee TR 45 Telecommunications Industry Association. Report to the federal communications commission on surveillance of packet-mode technologies, September 2000. http://www.tiaonline.org/policy/filings/JEM_Rpt_Final_092900.pdf.

[145] Council of Europe. Convention on cybercrime. Technical report, European Union, 2001. available from `http://conventions.coe.int/Treaty/en/Treaties/Html/185.htm`.

[146] Lambros D. Callimahos. *Traffic Analysis and the Zendian Problem*. Aegean Park Press, 1989.

[147] Michael F. Schwartz and David C. M. Wood. Discovering shared interests using graph analysis. *Communications of the ACM*, 36:78 – 89, August 1993.

[148] Joshua R. Tyler, Dennis M. Wilkinson, and Bernardo A. Huberman. *Email as Spectroscopy: Automated Discovery of Community Structure within Organizations*, pages 81 – 96. 2003.

[149] M. Rennhard and B. Plattner. Introducing MorphMix: Peer-to-Peer based Anonymous Internet Usage with Collusion Detection. In *Proceedings of the Workshop on Privacy in the Electronic Society*, November 2002.

[150] Krista Bennett and Christian Grothoff. gap – practical anonymous networking. In *Proceedings of the 3rd Workshop on Privacy Enhancing Technologies*. Springer, 2003.

[151] M. J. Freedman and R. Morris. Tarzan: A peer–to–peer anonymizing network layer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS2002)*. ACM Press, November 2002.

[152] Network Modeling and Simulation Project. Distributions of traffic stratified by application. Technical report, Cooperative Association for Internet Data Analysis (CAIDA), September 2002.

[153] KaZaA BV. `http://www.fasttrack.nu/`.

[154] `http://www.anonymizer.com/`.

[155] E. Gabber, P. Gibbons, Y. Matias, , and A. Mayer. How to make personalized web browsing simple, secure, and anonymous. In *Proceedings of Financial Cryptography 97*. Springer, February 1997. LNCS 1318.

[156] Ken A L Coar and D.R.T. Robinson. The WWW Common Gateway Interface, version 1.1. `http://cgi-spec.golux.com/draft-coar-cgi-v11-03.txt`, June 1999.

[157] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification. `java.sun.com/docs/books/jls/`.

[158] Netscape Communications Corp. Core JavaScript Guide 1.5. `http://developer.netscape.com/docs/manuals/js/core/jsguide15/contents.h%tml`, 2000.

[159] Virtuascape Media Design Agency. SWF format specification. `http://www.openswf.org/spec.html`.

[160] Microsoft ActiveX (TM) development kit. http://activex.adsp.or.jp/english/specs/overview.htm.

[161] World Wide Web Consortium W3C. HTML 4.01 specification. http://www.w3.org/TR/html4/, 1999.

[162] Andrei Serjantov, Roger Dingledine, and Paul Syverson. From a trickle to a flood: Active attacks on several mix types. In Fabien Petitcolas, editor, *Proceedings of Information Hiding Workshop (IH 2002)*. Springer-Verlag, October 2002. LNCS 2578.

[163] Inc. Doubleclick. Dart. http://www.doubleclick.com/us/corporate/privacy/privacy/internet-ads/da%rt.asp, 2003.

[164] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, 3 edition, July 2000.

[165] mod_perl team. mod_perl: Welcome to the mod_perl world. http://perl.apache.org/. Last Modified: Mon, April 21th.

[166] B. Kaliski and J. Staddon. PKCS#1: RSA cryptography specifications, version 2.0. IETF Draft, RSA Laboratories, September 1998.

[167] Sriram Srinivasan. *Extending Perl*, chapter 18. O'Reilly, 1997.

[168] Ian Robertson. Crypt::OpenSSL::RSA changes. http://search.cpan.org/src/IROBERTS/Crypt-OpenSSL-RSA-0.19/Changes, 2003.

[169] Openssl: The open source toolkit for ssl/tls. http://www.openssl.org. Last Modified: Thu, April 10th 2003.

[170] Steven M. Bellovin. Using the domain name system for system breakins. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 199–208. The USENIX Assoziation, June 1995.

[171] D. Eastlake. Domain Name System Security Extensions. RFC, Internet Engineering Task Force, March 1999. RFC 2535.

[172] D. Chaum and H. van Antwerpen. Undeniable signatures. In *Advances in Cryptology — CRYPTO'89 Proceedings*, pages 212–216. Springer–Verlag, 1990. LNCS 435.

[173] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, July 1970.

[174] Steven M. Bellovin. Using bloom filters for authenticated yes/no answers in the DNS. http://www.research.att.com/~smb/papers/draft-bellovin-dnsext-bloomfilt%-00.txt, December 2001.

[175] Ralph Merkle. Protocols for public key cryptosystems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, April 1980.

[176] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable certificate management using undeniable attestations. In *7th ACM Conference on Computer and Communications Security*, pages 9–18. ACM Press, November 2000.

[177] Ahto Buldas, Peter Laud, and Helger Lipmaa. Eliminating counterevidence with applications to accountable certificate management. *Journal of Computer Security*, 2002.

[178] Donald E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching, chapter 6. Addison–Wesley, 2 edition, 1998.

[179] Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03)*, pages 80–91, 2003.

[180] T. Pedersen. Noninteractive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology – CRYPTO'91*, pages 129–140. Springer, 1991. LNCS 576.

[181] Swox AB. Gnu multiple precision library. `http://swox.com/gmp/`, 2004.

Schlussfolgerungen

Teil 1: Ist Filtern überhaupt möglich?

In den letzten fünf Jahren gab es immer wieder Rufe nach Überwachung und aktiver Filterung des Datenflusses des Internets. Diese Rufe kamen von verschiedenen Seiten, manchmal den Medien (meist im Zusammenhang mit Kinderpornographie), Politikern (im Kontext ihrer jeweiligen Gegner), Geheimdiensten (im Zusammenhang mit höheren Budgets) und Medienkartellen wie der Record Industry Association of America (RIAA) oder der Business Software Alliance (BSA) (im Kontext von Copyright Verletzungen).

Diese Aufrufe an die Legislative nach Filtermöglichkeiten nehmen stillschweigen an, dass

1. Daten automatisch erkannt werden können, wenn sie in eine *semantisch*[3] definierte Kategorie fallen (wie etwa Kinderpornographie, politische Hetze, Anweisungen an Terroristengruppen, urheberrechtlich geschütztes Material).

2. die Verursacher die Filter nie umgehen können.

3. die kommunizierenden Parteien irgendwie identifiziert werden können.

Ob die Aufrufe zur zwangsweisen Filterung nun angemessen sind, und ob Filter wünschenswert sind [4] oder nicht, soll hier nicht diskutiert werden.

Der Hauptzweck dieses Teils der Arbeit war zu zeigen, dass selbst in sehr eingeschränkten TCP/IP–basierten Kommunikations–Netzen freie und anonyme Kommunikation möglich ist. In Kapitel 4 haben wir gezeigt, wie kurze Initialisierungs Nachrichten verschickt werden können so dass sie fast unmöglich zu blockieren sind, sobald Standard-Dienste des Internet zugänglich gemacht werden. In Kapitel 3 haben wir zahlreiche mögliche verdeckte Kanäle in Standard-Protokollen des TCP/IP Stacks aufgelistet, von denen einige durch Inspektion von Datagrammen nicht entdeckt werden können oder nicht blockiert werden können, ohne die Funktionalität des zugrundeliegenden Protokolls zu zerstören. In Kapitel 5 haben wird die Implementation eines verdeckten Mix-Netzes vorgestellt, die aussschliesslich auf dem all-

[3] und das auch noch sehr ungenau

[4] Wir bestätigen die Notwendigkeit von z.B. Spam–Filtern völlig, die in der Praxis den Irrtum hinter Annahme 1 oben zeigen.

gegenwärtigen Hyper-Text Transfer Protokoll beruht, und das unter normalen Umständen Unbeoachtbarkeit erbringt.

Teil 2: Schlussfolgerungen

Verteilte Datenbanken wie etwa DNS leiden an dem Problem, dass gekaperte Datenbank–Komponenten die Antworten böswillig verändern können. Daraus folgt, dass Überprüfung der Anworten gegen die Daten der Datenbank–Autoren nötig ist. Um diese Überprüfung möglich zu machen, können digitale Signaturen an die Einträge angefügt werden. Ein Problem bleibt übrig: die Überprüfung von "kein–solcher–Eintrag–vorhanden" Antworten, da es hier keine Daten zu signieren gibt und kein Autor verantwortlich ist. In diesem Teil der Arbeit haben wir einen effizienten Überprüfungsmechanismus vorgestellt, der dem Leser erlaubt, die Antworten einer Datenbank zu überprüfen, einschliesslich der Antworten, in denen die Datenbank die Existenz der gefragten Einträge verneint. Um dies zu erreichen, haben wir einen neuen kryptographischen Grundbaustein eingeführt, den Hash Baum mit Schlüsseln, eine Erweiterung des Merkle´schen Hash Baums. Unser Überprüfungsprotokoll ist effizient, es benötigt nur $2 \cdot |h()| \cdot |h()|$ Bytes zusätzlicher Daten für einem existierenden Eintrag und nur $2 \cdot \log(n) \cdot |h()|$ Bytes für nicht–exisitierende, wobei $|h()|$ die Länge des Outputs der Hash Funktion ist und n die Anzahl der Einträge in der Datenbank. Wir haben eine Speicherplatz–effiziente Datenstruktur zum Speichern und Verarbeiten des Hash Baums mit Schlüsseln auf Datenbankseite vorgestellt, zusammen mit Algorithmen zum Einfügen, Suchen und Löschen.

Wir haben die Beziehung zu Arbeiten im Gebiet Zertifikatsmanagement untersucht und gezeigt wie unser Ansatz auch dort benutzt werden kann. Im Vergleich zu den allgemeineren Zero–knowledge Sets von Micali et al. hat unser Ansatz den Vorteil, dass er schneller ist und nachträgliche Veränderungen durch mehrere Autoren erlaubt.

Lebenslauf

Geboren: 24. April 1971 in Neustadt a.d. Aisch

Grundschule: 1977 – 1981 Emskirchen

Gymnasium: 1981 – 1990 Friedrich-Alexander-Gymnasium, Neustadt a.d. Aisch

Studium: 1991 – 1997 Studium der Mathematik an der Friedrich-Alexander-Universität

Promotion: 1997 – 2004 Promotion am Institut für Informatik der Friedrich-Alexander-Universität