ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Adrian von Bidder

Key Exchange (KX) – A Next Generation Protocol to Synchronise PGP Keyservers

16.7.2003
Semester Thesis SA-2003.04
Winter Term 2002/2003

Supervisor: Nathalie Weiler
Professor: Bernhard Plattner

Abstract

In the Internet, securing email has always been an important issue. Various standards and products have been created. One of the most successful standards is OpenPGP, which uses public key cryptography (RSA and others) and is implemented in systems like Pretty Good Privacy, GNU Privacy Guard, Hushmail and others.

A well-known difficulty with the use of public key cryptographic systems is the verification and distribution of the public keys. OpenPGP solves the problem of verifying the authenticity of a public key by having users certify each others keys, building a "Web of Trust" by bundling these key certificates with each users public key. Therefore, adding a new public key and updating an existing public key (or replacing it by a new version) are the two most important operations of any PGP public key repository.

To allow easy distribution of PGP public keys, the OpenPGP community established a network of open access public keyservers, allowing users of OpenPGP software to freely exchange public keys. The nodes of this keyserver network synchronise their database by exchanging new public keys and key updates amongst each other, virtually building one global key database. At the moment, this synchronisation is done with an inefficient and ineffective email based protocol. This semester thesis describes the implementation of an alternative protocol – KX – based on direct TCP connections between the keyservers and unambiguous identifiers for every key update or new key. By dropping the dependency on a working mail system and using improved error halndling mechanisms, KX is a lightweight alternative in terms of used network, disk and CPU resource useage.

Zusammenfassung

Die Sicherheit von Email-Kommunikation war im Internet schon immer wichtig, daher gab und gibt es diverse Lsungen fr dieses Problem. Einer der am weitesten verbreitetsten Standards ist die Public Key Verschlsselung mit OpenPGP, implementiert in Produkten wie PGP, GPG, Hushmail etc.

Zwei der wichtigsten Probleme im Gebrauch von Public-Key Kryptosystemen sind die Verteilung und die Verifikation der Public Keys. OpenPGP authentisiert die Public Keys, indem jeder Benutzer die Keys von andern Benutzern berprft und unterzeichnet und so deren Echtheit besttigt. Diese Unterschriften werden dann mit den Public Keys zusammen verteilt und bilden ein “Web of Trust”. Das hinzufgen von neuen Public Keys und das ergnzen eines Public Keys um neue Unterschriften sind daher die wichtigsten Operationen auf jedem Public Key Repository.

Um einfachen Zugang zu OpenPGP Public Keys zu ermglichen besteht ein Netzwerk von PGP Public Key Servern im Internet, die gegenseitig neue Keys und Unterschriften austauschen und so eine globale Public Key Datenbank darstellen. Das gegenwrtigverwendete Synchronisationsprotokoll zwischen diesen Servern ist recht ineffizient und soll durch das in dieser Semesterarbeit vorgestellte Protokoll – KX – ersetzt werden. Die Email-basierte Kommunikation wird dabei durch direkte TCP-Verbindungen zwischen den Keyservern abgelst, was bessere Fehlerbehandlung sowie Einsparungen bei der CPU, Disk- und Netzwerkbelastung erlaubt.

Preface

Computer security, and in particular email security and PGP (Pretty Good Privacy) have always interested me. Writing a semester thesis on this topic was therefore quite the obvious thing to do, especially when I met Nathalie Weiler and learned that she had been involved with running the Swiss PGP keyserver for a time.

Shortly after I started working on this semester thesis in October 2002, Nathalie asked me if I would present my work as a conference paper at the "IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises"[1] (WETICE), so the main part of this report is the paper we submitted for the workshop (which subsequently was accepted and which I presented successfully at the conference in Linz, Austria). The appendices contain some more detailed implementation notes which I couldn't include in the WETICE paper because of space constraints and because WETICE has too broad a scope for the audience to be interested in these details.

Finally, I'd like to thank all the people who made this project possible. Of course, it is impossible to list everybody by name, but I'd like to specifically mention Patrick Feisthammel, Diana Senn, Marcel Waldvogel, Jason Harris and Richard Laager (no particular order). And of course, I want to thank Prof. Bernhard Plattner and Dr. Nathalie Weiler for supervising me and being infinitely patient regarding my rather relaxed attitude towards the original schedule for this project.

[1] http://wetice.jpl.nasa.gov/wetice03/; in case this link is not valid: the conference proceedings will be archived in the IEEE digital library at http://ieee.org.

Contents

Key Exchange (KX) – A Next Generation Protocol to Synchronise PGP Keyservers

Adrian von Bidder and Nathalie Weiler
Swiss Federal Institute of Technology ETH Zürich, Switzerland
avbidder@fortytwo.ch|weiler@acm.org

Abstract

In the Internet, securing email has always been an important issue. Various standards and products have been created. One of the most successful standards is OpenPGP [4], which uses public key cryptography (RSA [13] and others) and is implemented in systems like Pretty Good Privacy [15], GNU Privacy Guard [8], Hushmail [1] and others.

A well-known difficulty with the use of public key cryptographic systems is the verification and distribution of the public keys. OpenPGP solves the problem of verifying the authenticity of a public key by having users certify each others keys, building a "Web of Trust" [5] by bundling these key certificates with each users public key. Therefore, adding a new public key and updating an existing public key (or replacing it by a new version) are the two most important operations of any PGP public key repository.

To allow easy distribution of PGP public keys, the OpenPGP community established a network of open access public keyservers [7], allowing users of OpenPGP software to freely exchange public keys. The nodes of this keyserver network synchronise their database by exchanging new public keys and key updates amongst each other, virtually building one global key database. At the moment, this synchronisation is done with an inefficient and ineffective email based protocol. This paper describes the implementation of an alternative protocol – KX – on the popular `pksd` *keyserver [6], based on direct TCP connections between the keyservers and unambiguous identifiers for every key update or new key. With the dropping of the dependency on a working mail system and the improved fault mechanisms, KX is a lightweight alternative in terms of used network, disk and CPU resources.*

Keywords: *OpenPGP, Keyserver, Secure Synchronisation Protocol, E-Mail Security.*

1 Introduction

With the advent of protection mechanisms for e-mail such as PGP (Pretty Good Privacy) [15] or S/MIME (Secure Multipart Message Exchange) [12] based on public key cryptography, a need arose for a method to exchange and distribute the necessary public keys. While S/MIME relies on the existence of a public key infrastructure (PKI), PGP uses a world wide distributed keyserver network: the public keys are *published* on a keyserver, i.e. they are collected on a keyserver where they can be retrieved by the interested user. In order to provide a useful service, the keyservers synchronise their public key databases among themselves.

The synchronisation protocol has seen little evolution over the last years since its development by Marc Horrowitz. Basically, every new PGP public key or new signature on a PGP public key triggers a flooding of the information through the keyserver network. Although new synchronisation protocols using multicast [3, 14] or a whole new approach to build a keyserver [9] have been proposed, the authors did not succeed to get their solutions deployed in the current keyserver network. The main reason was a reluctance of the around thirty keyserver maintainers to switch to whole new keyserver untested under normal operation. Additionally, the lacking backward compatibility made it difficult to make it a convincing alternative. So, instead of getting rid of the cumbersome, error prone original synchronisation protocol, the maintainers stick to the old system.

In this paper, we present an alternative approach to a better synchronisation protocol for the keyserver network. Our *Key Exchange KX* achieves a significant reduction of the flooding with less errors than the original protocol while still sticking to the "old" keyserver software and network structure. KX needs not be run by all keyservers, making a smooth transition possible and guaranteeing backward compatibility. Thus, we believe that KX will be easily adopted into the current keyserver network – a belief confirmed by private communication with several keyserver maintainers.

In the remainder of the paper we will first describe the shortcomings of both the current keyserver software and the synchronisation protocol used among the keyservers, and we will discuss related work in Section 2. Section 3 details the design of our protocol KX and Section 4 describes the implementation. Finally, Section 5 compares KX to the existing protocol. We conclude with the further implementation plan into the operational keyserver network in Section 6.

2 The Past and the Present

This section describes the various existing solutions for exchanging OpenPGP public keys between keyservers.

2.1 Key Exchange by Email

The key propagation protocol defined by Marc Horowitz' `pksd` keyserver software is the only one in widespread use today ([6, 2]). The protocol synchronises the keyservers by distributing key updates via email using a relatively simple flooding algorithm:

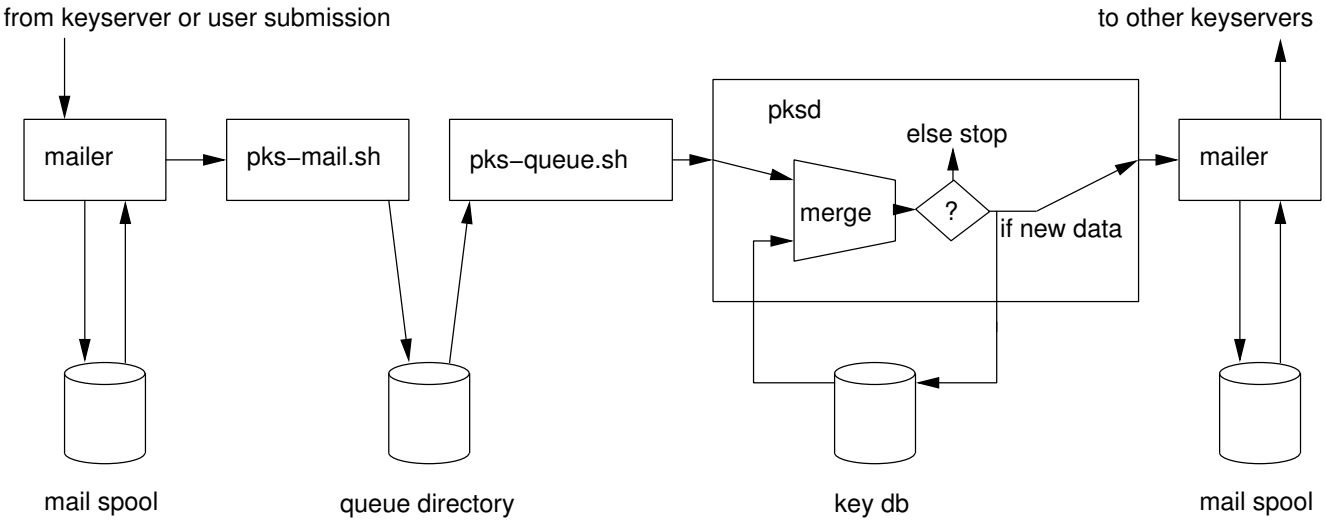

Figure 1. Flow of a key update (or new key) through pksd.

Every keyserver configuration contains a list of peers that should receive a key update message (*INCREMENTAL*) whenever a key changes or a new key is submitted (by a user submission, or by an incoming key update message). To avoid excessive redundancy, `pksd` places *X-Keyserver-Sent* headers in the email messages, and does not resend a key update to servers that were already listed in the incoming message.

Because this protocol is currently in widespread use, its problems are well known[1]:

Duplicates: Keyservers with many peers will often receive the same update message over and over again, despite the X-Keyserver-Sent header. While obviously only the first message changes the key and causes further update messages to be sent, every update message has to be processed fully, a waste not only of network bandwidth, but also of considerable CPU power. The major problem is that mail delivery is quite resource intensive in terms of process creation and disk I/O. This is easily seen in Figure 1: the mail transport system spawns a `pks-mail.sh` shell script for every new message which writes it to a queue directory. The `pks-queue.sh` script picks this message up and, using the `pksdctl` command, feeds it to the keyserver process. Outgoing update messages are again processed by the local mail system which usually writes the message into a spool directory before sending it off to the target mail system.

Unreliable transport: When an email fails to be delivered, a delivery failure notification is sent back to the originator of the message. But as these reports are not standardised, it is very difficult to track which update message failed. So, `pksd` does not use these error messages at all and thus does not detect lost updates (due to keyserver, mail server or network outages). The result is a divergence of the keyserver databases over time.

No flow control: The problematic error reporting of email delivery failures is also the cause of `pksd's` bad behaviour on network or server outages: `pksd` will continue to send email to dead (or temporarily offline) remote servers, causing mail messages to pile up on intermediate mail servers. Once the remote server comes online again, it is confronted with a flood of incoming messages, causing load problems and usually requiring manual intervention.

2.2 Using Multicast

The solutions proposed by Baumer [3] and Waldvogel, Mohandas, Shi [14] use multicast to distribute key updates and new keys in a more efficient manner. Baumers *Binary Keyserver Protocol* was implemented as an addition to `pksd` and used IP Multicast, while the *Efficient Keyserver using ALMI (EKA)* is an independent keyserver implementation and uses the application level multicast library ALMI [11]. The idea behind both proposals is that key updates should be identified by a globally unique number, which is generated by combining a (globally unique) issuer id and a (local) serial number. The state of a node thus consists of the highest serial number seen from each issuer.

In addition to the problems of unavailability of IP multicast in large parts of the Internet (a problem for BGP) or the dependency on a Java environment and a relational database system (as with EKA), both approaches suffer from the need to track all other keyservers. Once keyservers become very common, synchronising the server status requires exchanging serial numbers of possibly thousands of

[1] The `pksd` keyserver also has some other shortcomings, e.g. corruption of some keys in certain versions; these are not discussed here.

issuers, and a malicious attacker might be able to introduce millions of bogus issuer IDs which would spread over the whole network.

2.3 Set Reconciliation

The *Synchronising Keyserver* [9] uses a completely different approach to the problem of synchronising keyservers. Instead of solving the problem of reliably distributing new keys and key updates, it uses a set reconciliation algorithm [10] to synchronise the key databases directly, with communication costs being linear in the number of differences between the databases, independently of the database size.

SKS is actively developed, and some of its initial problems – such as the inability to synchronise with the email based `pksd` keyserver network – have been solved or are being solved.

Currently, the main problem with SKS is that it is very new and not in widespread use, and therefore not as well tested as the current `pksd` keyserver. Thus, the keyserver maintainer are reluctant to change to SKS[2].

Also, with its focus on reconciling whole databases, setting up partial keyservers (like keyring.debian.org, carrying only keys of Debian developers – it does not allow new keys to be added, but it allows updates to existing keys) still require status information about the full database (this status information is much smaller than the actual key data, though.)

3 Our Solution

Our proposed protocol must, of course, solve the main problems of the current email based scheme: (1) the high processing cost of redundant update messages and (2) the bad behaviour on errors. We do not address most issues that can be attributed to implementation problems with the current `pksd`, specifically the issues with the database code, as this is out of the scope of this paper.

3.1 The Basic Idea

The fundamental idea of the proposed protocol is that new key data is exchanged on the keyserver network in two stages: first, the availability of a new key update[3] is announced, and only in a second stage the actual key data is distributed. Processing these announcement in the first step is cheap in both communication and processing cost – just check if a particular update has already been received. Only in a second stage, the full update message with the key data is transmitted and processed. The configuration of the keyserver network has not changed from the current protocol: the list of neighbours is manually configured at each keyserver[4].

Since we focus on reliably distributing key updates, as the current protocols [14, 3] do, a way to unambiguously identify a key update is needed. We have shown in Section 2.2 that using global version numbers with an issuer ID is problematic in an open network like the keyserver network. KX solves this problem by using a *hash value* over the key data (we'll discuss details in Section 4.2) to identify key updates, so that – if the current keys in the key databases are identical – the identical update will be generated regardless of the location of injection[5]. When key updates are announced, this hash value is used to identify each update message. Additionally, the fingerprint of the affected key is announced and each update is assigned a local sequence number.

To keep track of key updates, it stores the highest sequence number of each neighbour to make sure that it hasn't missed an update, and – when new updates are announced – requests updates with previously unseen hash values, while ignoring those with hash values it has already seen.

Since this two step process is somewhat asynchronous – after an update message has been announced, other keyservers may want to retrieve it at any time – keyservers using KX are required to store the update message for at least 72h after the initial announcement (this value was chosen arbitrarily, real world experiences may show that a smaller value is sufficient, or that a bigger value is necessary.)

3.2 Protocol design

The core of KX consists of four basic messages, with in a few additional messages (such as CLOSE and ERROR) being specified but not discussed in this document. Likewise, the exact syntax of the messages is not discussed here – the protocol specification will be contained in another document which is yet to be published.

LIST requests a directory of available key updates (within a specified range of sequence numbers) from the peer, which responds by sending a corresponding IHAVE message.

IHAVE announces the availability of one or multiple update messages. Each update message is specified with its sequence number, fingerprint of the affected key, and the hash value over the whole message. Note that IHAVE is used both as answer to a LIST request and spontaneous to notify keyservers when a new update is present.

GETUPDATE requests a specific update message, identified by its sequence number.

UPDATE delivers an update message as requested with GETUPDATE. For easy debugging, key data is exchanged in ASCII armored form. If network bandwidth proves to be an issue, a next protocol version might want to switch to binary format.

The typical pattern of interaction between neighbours using KX is shown in Figure 2: When the keyserver is notified

[2] We believe that our approach will be given a better acceptance in the community, because we "only" propose replacements for malfunctioning parts in the keyserver and not a whole new design. The past shows that none of the new designs [3, 14] were adopted for similar reasons by the community.

[3] The terms *key update* or *update message* as used in this paper refer to both a new key or an update to an existing key.

[4] The graph of the current keyserver network can be retrieved at on `http://www.rediris.es/keyserver/graph.en.html`.

[5] A mechanism to reconcile key databases in the cases where differences have accumulated in the past is being thought about.

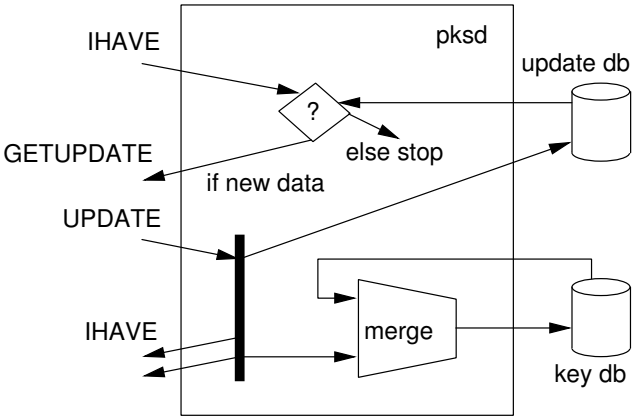

Figure 2. Exchange of key updates with KX.

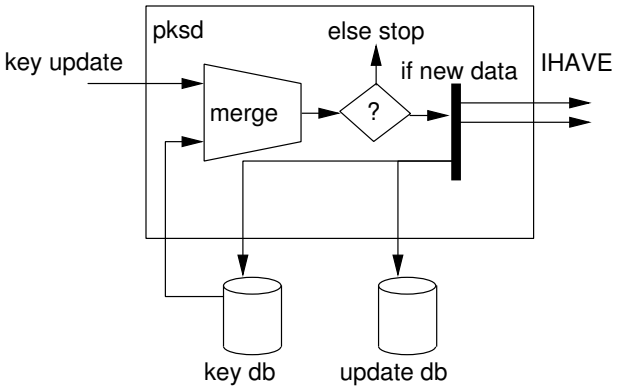

Figure 3. Interaction of KX with legacy protocols.

by a neighbour that a new key update is available, it looks in its database of recently received key updates whether an update with the same hash value has already been received. If so, no further action is necessary. If the update is new, a GETUPDATE message is sent to the neighbour who has announced this update. In the current implementation, this happens immediately. As keyservers are required to store key updates for at least 72h, a keyserver can send this request at a later time, for example when the network load is lower. When the UPDATE is received, the keyserver assigns it a local sequence number, stores it in the update database and announces it to all other neighbours. Of course, the update is also merged into the key database of the keyserver.

Storing the incoming update directly into the update database (instead of merging it into the key database first, and regenerating the update from the results of the merge) has the advantage that the key distribution mechanism does not depend on particular features or a particular version of the key merging code – for example, when keyservers with support for photographic userids on public keys become available, the photoid will be distributed in the keyserver network even when most keyservers cannot use this data yet. The downside is, of course, that bogus data created by some buggy keyserver implementation is propagated through the whole network and not filtered at the first hop.

Another type of interaction occurs on connection setup: when a new connection between keyservers is established, both keyservers spontaneously announce the highest available sequence number in an IHAVE message. If a keyserver detects that it is missing some updates, it requests the directory of the missing updates with a LIST message, which again gets answered by an IHAVE message. From this point, processing is the same as above.

Interaction with the world outside of KX – with users submitting new or updated keys, or with the legacy pksd protocol – is shown in Figure 3. A update message is generated from the output of the key merger (this is the same message that is sent out in the email based pksd protocol, then stored in the update database and announced with IHAVE messages to the other keyservers.

4 Implementation

KX has been implemented on top of Marc Horowitz' pksd [6] – a project continued on Sourceforge [2]. The current version (0.9.6) has been released in February 2003, with frequent new releases correcting many long standing problems in the near future.

One aim with the design and implementation of KX has been to avoid any additional dependencies on external libraries – this is the main reason why the protocol is a simple line oriented text protocol and does not use a standardised (and more generic) network protocol, for instance the XML based SOAP protocol or the Network News Transport Protocol (NNTP), as has been suggested in discussion on the `pgp-keyserver-folk@flame.org` mailing list.

The implementation is entirely done within the one-process model of pksd – no additional process is created, and no multi threading is introduced. The two major building blocks are the update database and the algorithm for the update hash. The third one is the state machine to handle the protocol, which, in this implementation, is quite simple: all activities are carried out immediately, no batching or scheduling of activities to times of low activity is being done. This is purely a property of this implementation, and does not result directly from the protocol specification.

4.1 Update Database

As the update database is relatively constant in size and not really big (a few thousand update messages) but with content that totally changes after approx. 72h, because old entries are expired, a very simple design has been chosen: each update message is stored in a plain file, with the associated hash value as filename. Additionally, there is an index file to allow access to the updates by the sequence number. Expiry of old entries is done by a simple cron job that just removes the files older than 72h.

The update database is accessed in the following ways as messages come in:

- The processing of LIST messages needs read access by sequence numbers.

- IHAVE messages result in a stat(2) system call, checking if an update with a specific hash value (the filename of the update message) is already known in the database.
- GETUPDATE messages cause an update message (identified by its sequence number) to be read from its file.
- An UPDATE message causes a new file to be written and inserted into the index file with the next available sequence number.

4.2 The Update Hash

The definition of an unambiguous hash value over an OpenPGP public key is probably the single most important detail in the specification of KX. At this point it is necessary to delve into the format of the key data, as specified in [4]. The RFC specifies a partial order of key packets: primary key and direct key signatures, then userids with their signatures, and subkeys with their signatures at the end. An order between signatures belonging to the same userid or subkey, or between userids and between subkeys, is not defined, though. So, for the purposes of hashing the key data, the packets are ordered by directly comparing the binary data of the key packets, without its length specifier. This respects the constraints given by the RFC, a key with packets ordered by this algorithm is still a valid OpenPGP key. The length specifier is being left out because the same length can be encoded in different ways – see sections 4.2.1 and 4.2.2 of [4]. Ordering by binary comparison (memcmp) has been chosen because it's efficient and does not require any parsing of the data, and because the ordering has no significance outside of KX, so there's nothing to be gained by using a more 'sensible' approach like ordering by keyid, time stamp, or userid.

Because the current KX implementation does not change an update message after it has been generated, the ordering step could have been omitted altogether. But the goal was to generate the same update hash if two updates contain the same key packets, regardless of the state of the database at the server issuing the update. This also allows KX implementations in which update messages are not explicitly stored outside the key database, but are regenerated on the fly.

5 Evaluation

This section compares KX with the classical email based key distribution method (referred to as the *pksd protocol* here for lack of a better name). No benchmarking of other key servers with their protocols (EKA [14], SKS [9] or BGP [3]) could be done, as these are not in widespread enough use today that significant data could be gained.

5.1 CPU and Network Costs

The biggest performance gain between KX and the email based protocol originates from the difference in the used *transport mechanisms*: pksd sends key updates per email, and this means that for every message a number of processes and at least one TCP connection are created (as already introduced in Figure 1). In KX, direct and persistent TCP connections between the pksd processes are used – this means that no process and at most one TCP connection (in the unlikely event that the connection has broken down or timed out) has to be created. Also, since no mailer is involved, disk I/O from and to spool directories does not occur outside of pksd.

Another problem with the pksd protocol is the *number of duplicated messages*: An analysis of 10 days of the server log files of `wwwkeys.ch.pgp.net` showed more than 40000 incoming 'incremental' key updates, but only 9000 of these contained new key data and thus caused update messages sent to the other key servers. In KX, we expect a similar ration between the IHAVE and UPDATE messages, the big difference is that an IHAVE message is much cheaper to process: the message itself is smaller, and the decision if a GETUPDATE/UPDATE step is necessary does not require much computation – with the current implementation, the check if this update has already been seen is done by checking if the corresponding file does exist.

5.2 Error behaviour

The mail based pksd protocol does simply not handle transmission errors - email bounces are not handled, so the keyservers do not notice lost key updates and the key databases diverge over time. Also, when a server (or its Internet connection) is down, its neighbours will continue to send email and causing update messages to pile up on intermediate mail servers, possibly resulting in mail server problems, and in any case causing high network and CPU load spikes when the server comes back online.

The use of direct TCP connections between the keyservers allows the implementation to detect these problems easily. On most problems, our implementation of KX just closes the connection. When the next update has to be announced, the connection will be reestablished, and during the connection set up, the two servers will detect if any key updates failed to transmit and request these again – there is no retransmit mechanism; on the public keyserver network, key updates happen frequently enough, so this should not be a problem. On a private (corporate) keyserver network, this part of the protocol implementation probably needs reexamination. If a key update is not available anymore, the administrator of the keyserver is notified (through the usual syslog mechanism) and will have to take action manually, either examinating the server log files on the sending server or by using pksd's *LAST* command (returns keys modified in the last *n* days). As stated above, this should only happen if a server has been unable to retrieve a key update for more than 72h.

5.3 Security considerations

Currently, the KX protocol operates on unencrypted TCP connections, and authentication is only done on the basis of IP addresses, accepting only connections from configured, i.e. known peers. So, our KX implementation is vulnerable

to IP spoofing attacks. While this could (and should) be corrected (by using authenticated and encrypted connections, with SSL for instance) in a future version of KX, we do not regard this as a serious flaw, since the possible damage by injection of malicious data is only minor[6]. Currently, in an IP spoofing attack, a malicious attacker could inject arbitrary IHAVE messages (with sequence numbers, update hashes and key fingerprints) or arbitrary key update messages, and suppress the circulation of a certain amount of upgrade messages:

(1) *Injecting a very high sequence number* appearing to come from some keyserver K can cause the target keyserver to ignore some key updates in some circumstances. In most cases, however, when K announces the next regular update (with a low sequence number), the target keyserver will detect that the sequence number must have wrapped around on the 31 bit boundary and will just request all available updates from K (as it has apparently missed the updates between the spoofed, high sequence number and the current one). As this can be done with only a few small messages, it is not even a strong Denial of Service attack - at most, all stored update messages are transmitted in one batch, which could block a server for a few minutes.

(2) An attack on keyservers by *injecting arbitrary update data* is equally possible with todays email and web frontends, and so it is not discussed here.

(3) By inserting a well designed and well timed UPDATE message in a KX connection between two keyservers, an attacker can *suppress one update*. If the attacker is in a position to do this for all connections leading to the target keyserver, the update can be suppressed totally – but in that case we must assume that the attacker is very close to the keyserver and has total control over the network, and could do much worse. If the attacker can do this only on one connection, this attack will be harmless if the target keyserver is directly connected to more than one neighbour, as it will retrieve the same update from a different neighbour.

While this section certainly is no formal analysis of the security of the KX protocol and its implementation, we believe that the protocol should be robust enough for use in the public keyserver network.

6 Conclusion

In this paper, we described a synchronisation protocol for the popular PGP keyserver network. The current protocol has some serious performance drawbacks resulting in a lack of global sycnhronisation of the PGP public key database. KX is a new design and implementation of an alternative protocol for the popular `pksd` keyserver. It concentrates on reliability and scalability and replaces the email based transport mechanism by direct communication between the keyservers through TCP connections. It uses a two stage mechanism: first it announces the availability of new data, and only in a second step, and only when necessary, is the actual key data retrieved. By this approach, we could significantly lessen the CPU, disk and network load caused by key synchronisation.

The implementation is currently working, and we are planning to (1) further test it, (2) integrate it with the official `pksd` sources [2] and (3) have it actually used in the public keyserver network, so the email based protocol can finally be retired. Naturally, it will be important to maintain the implementation as `pksd` evolves or new bugs are discovered.

[6] By minor we mean: restricted to only few updates or only few keyservers. Of course, one single missed revocation certificate can be disastrous to a key user, but it doesn't affect the keyserver network.

Acknowledgments

The authors would like to thank the keyserver maintainers (`pgp-keyserver-folk@flame.org`) for their valuable feedback and support on our approach. A special thanks goes to Patrick Feisthammel for the live data used as basis in Section 5.

References

[1] Hushmail. `https://www.hushmail.com/`.

[2] OpenPGP public key server. `http://sourceforge.net/projects/pks/`.

[3] M. Baumer. Distributed server for PGP keys synchronized by multicast. Semester thesis summer term, 1998. `http://www.tik.ee.ethz.ch/tik/education/sadas/SASS1998-33/thesis.ps.gz`.

[4] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP message format. RFC 2440, November 1998.

[5] P. Feisthammel. Explanation of the Web of Trust of PGP. `http://www.rubin.ch/pgp/weboftrust.en.html`, 1997.

[6] M. Horowitz. PGP public key server, 1997. `http://www.mit.edu/people/marc/pks/`.

[7] PGP keyserver network. `http://www.pgp.net/pgpnet/`.

[8] W. Koch. GNU privacy guard. `http://www.gnupg.org/`.

[9] Y. Minsky. SKS: Synchronizing key server. `http://sks.sourceforge.net/`.

[10] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. In *International Symposium on Information Theory*, 2001.

[11] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 49–60, March 2001.

[12] B. Ramsdell. S/MIME Version 3 message specification. RFC 2633, June 1999.

[13] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[14] M. Waldvogel, R. Mohandas, and S. Shi. EKA: Efficient keyserver using ALMI. In *Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2001.

[15] P. Zimmerman. *The Official PGP User's Guide*. MIT Press, Boston, U.S.A., 1995.

Appendix A

The KX Protocol

Where the conference paper could only give a brief overview of the KX protocol messages, this section will provide more details on what these messages actually look like on the network.

A.1 General Message Format

While the protocol uses two way communication between the keyservers, the communication is completely asynchronous and can without problems be implemented on top of a packet oriented transport layer. Also, the protocol specification itself does not require an error correcting, lossless transport protocol, although the current implementation is only available on TCP.

Messages are human readable to ease debugging (encoding overhead is not a problem with key updates, so a more compact binary message format does not buy any great advantage). All messages are as self contained as possible, to minimize the state to be kept on the keyservers. The general schema of a message is

```
<tag>
<command>
<zero or more lines of data>
.
```

with the usual CRLF sequence as line separator, and commands never having any arguments (on the same line, that is). Tag is any sequence of 4 printable ASCII characters and is used to associate responses to requests. Of course, tags have to be unique for each session (for connection oriented transport mechanisms) or at least within a reasonable timeframe.

The data is also in ASCII form whenever possible (right now: always). For comands that are interepreted as replies to previous requests, the first line of data is the tag of the associated request.

A.2 The KX Protocol Messagers

As the four basic messages (`LIST`, `IHAVE`, `GETUPDATE` and `UPDATE`) are already described in the WETICE paper, only the syntax of these messages is described here. More space is dedicated to the `HELO`, `ERROR` and `CLOSE` messages.

A.2.1 LIST

Request a listing of available updates. Each data line is a sequence number or a range of sequence numbers (`<low>-<high>`), potentially open-ended.

A.2.2 IHAVE

Is sent in answer to a LIST request (first data line is the tag of the `LIST` request) or spontaneously (first data line is `NOTIFY`) when a new update becomes available. From the second data line: `<sequence number> <key fingerprint> <update hash>` (but see the notes in the next chapter about including the key fingerprint).

A.2.3 GETUPDATE

Data is one sequence number per line.

A.2.4 UPDATE

First line is the tag of the respecitve `GETUPDATE` message, second line is the update specification as in the `IHAVE` command, third and subsequent lines contain the OpenPGP key material in ASCII armored form.

A.2.5 HELO

The `HELO` message is used as a greeter and is sent from both ends of the connection as soon as the connection is set up[1]. As soon as each end of the connection receives the peer's `HELO`, an `IHAVE` message is sent announcing the newest available key update.

The `HELO` message has one data line, the version of the protocol, which is set to `1.0` in this specification.

A.2.6 ERROR

The `ERROR` message carries three data lines: the tag of the offending command (or `NOTIFY` if the error was not caused by a command[2]), a numerical error code and in the third line a short human readable error message.

In the current implementation, the following error codes are used:

500 Error code 500 is used when a generic error is encountered, where it is not expected that a peer can handle the error condition sensibly anyway. Usually, closing and re-establishing the connection is the only possible solution.

501 *Problem handling this error* When a previous error message could not be handled by the recipient, this error is sent back. This will usually be followed immediately by a CLOSE message and breaking the connection. Normally, error messages

551 *Command sequence error* At command startup, each participient in the cnnection expects an `HELO` – `IHAVE` sequence from the other side. If this protocol is not followed, this error is sent.

A.2.7 CLOSE

When one side of a KX communication wants to close down, it sends a `CLOSE` message without any data, which is then answered by a `CLOSE` message of the other party with the tag of the first message as the only data line, before the connection is effectively tore down.

Implementations may chose (as the current implementation does) not to send close messages at all but to close the connection immediately, when the transport mechanism is able to deliver an indication of when this is the case. In any case, an implementation must be prepared to receive and correctly handle `CLOSE` messages.

[1] If the protocol is used in a connection-less setting, the sections regarding the `HELO` and `CLOSE` message will have to be revised.

[2] This is currently not used

A.3 Example Conversation

The following is an example message showing a connection set up between two keyservers. It is assumed that the 'right' keyserver is up to date while the 'left' keyserver (sending everything prepended with `>`) missed everything after the `IHAVE` message with the sequence number 40. However, it recieved the actual update data of most missed `IHAVE`s from some other source and thus only needs to retrieve the update 43 from on this connection.

Note, too, that for better readability the process is shown as a proper dialogue. In practice, the two half-connections are completely independent, and the keyservers don't wait until the remote end has finished sending a message before sending its own message.

```
< 0001
< HELO
< 1.0
< .
> 100a
> HELO
> 1.0
> .
< 0002
< IHAVE
< NOTIFY
< 45 EFE396F418F58D65849428FC1438516892082481 59f02366fe4cb14d3af6cf4e01d2f998
< .
> 100b
> IHAVE
> NOTIFY
> 190 97FC36318F175FBA6CF28FEF33F1B866BE769BDF 34d6e61ce50208d01723cfa61965b0e4
> .
> 100c
> LIST
> 41-
> .
< 0003
< IHAVE
< 100c
< 41 3E200377663F6B09E474FDAE801568FC 9b4f7cbf22b6def1f8d579b969fd4d58
< 42 DCAF16D5BB56B970A8F9EC88E4CE6199D0980A99 762d242bf97d3e5e2f5a269e7b3c661d
< 43 11C294DF1D6C9698FEFE231D3BF609C68BAFCDBD 0beaa9e85391ecad4ebf1a9ce9d51455
< 44 96A65F711C438423D9AE02FDA71197BA 4d85fbcf7c1360fc59c35f4b8513cf7e
< 45 EFE396F418F58D65849428FC1438516892082481 59f02366fe4cb14d3af6cf4e01d2f998
< .
> 100d
> GETUPDATE
> 43
> .
< 0004
< UPDATE
< 100d
< 43 11C294DF1D6C9698FEFE231D3BF609C68BAFCDBD 0beaa9e85391ecad4ebf1a9ce9d51455
< -----BEGIN PGP PUBLIC KEY BLOCK-----
< Version: PGP Key Server 0.9.5
<
< mQGiBDsdr9ARBADdPp6U5CAo9VuohkkdNonZ77VO0Hd+Dr1Jh5yMVH9hSW6FsXfj
```

```
< 5c1/S0+77sXJw1GpayoUkqOqP8SAKJxyP2bNAIoeB71b1IxczXiKUSWgPQhrDz4r
  ...
< iisP9jMcgcYRMIhGBBgRAgAGBQI7Ha/UAAoJEDv2CcaLr829b7IAoNdYzJdNtzbm
< pOMHf5xxeCFSOVBxAJ96i8ES0jzmbN+r2jWgPpL1dUwWZw==
< =W4X1
< -----END PGP PUBLIC KEY BLOCK-----
< .
```

As soon as the two first `IHAVE` messages are received, the connection is initialized and will be used regularly to exchange key updates with `IHAVE` (`NOTIFY`) messages (and successive (`GET`)`UPDATE` messages, if necessary).

Appendix B

Implementation

This project was started shortly after Richard Laager put the pksd source code into the repository on sourceforge[1] and collected the various patches, releasing pksd version 0.9.5. The project cvs repository on the CD-ROM contains copies of the upstream code on the `sourceforge` branch, with cvs_yyyymmdd tags. At the end of my work on the KX project, pksd 0.9.6 was out but the sourceforge cvs had already developed quite a bit further.

The major part of the code is in the three new files `pks_sync.c`, `sync_cs.c` and `sync_db.c` and the associated header files, with only minor modification to the existing code. The following sections give a file-by-file overview of all code modifications done for KX.

B.1 `mail_req.c` and `pks_www.c`

These files contain the web and mail frontend to pksd. A call to `pks_sync_post()` was added, to send out KX messages when a new update is received by the email or web frontend.

B.2 `multiplex.c`

The multiplexer listens on open connections with the Posix `select(2)` system call, and uses callback functions to communicate with the other program modules. The `mp_add_write_again()` function was added. It buffers data to be written to a filedescriptor where some data is already waiting.

B.3 `pks_config.c`

The configuration file parser was extended to recognize the new `tcp_syncaddr`, `tco_syncport`, `tcp_syncsite` and `sync_db_dir` directives KX uses. Also, the `log_terminal` directive is new, used to direct all logging information to stdout instead of syslog to aid debugging.

B.4 `pks_sync.c`

This new file contains the implementation of the KX protocol: the `execute()` function to interepret and act on incoming messages, and the `pks_sync_post()` function to announce new updates to the other servers. The database to track the sequence numbers of all neighbours (`tcp_syncsite`s) also is in this file.

[1] http://pks.sf.net

B.5 `sync_db.c`

Also a new file, it contains the database of key updates. In this implementation, key updates are stored as files in a directory, with the MD5 hash of the update as filename. Additionally, there is an index file mapping sequence numbers to filenames.

B.6 `sync_cs.c`

The third new file contains a simple OpenPGP key parser, and the sorting algorithm which defines the canonical ordering of a key to determine the update hash value.

Institut für
Technische Informatik und
Kommunikationsnetze

Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Winter 2002/2003

Semesterarbeit

für

Herrn Adrian von Bidder (D-INFK)

Betreuerin: Nathalie Weiler

Ausgabe: 1. Oktober 2002
Abgabe: 1. März 2003

A next Generation Syncronisation Protocol for PGP Keyservers

1 Einführung

Pretty Good Privacy (PGP) ist eine auf hybrider Verschlüsselung basierende Software, die hauptsächlich im Bereich der authentischen und vertraulichen E-Mailübertragung eingesetzt wird [Sta94, CDFT98]. PGP hat sich auf dem Markt etanliert als ein Open Source Produkt, welches die geheime und authentische Übertragung von Daten ermöglicht. Der Erfolg von PGP basiert auf seinem dezentralen Schlüsselmanagement und dem imhm zugrunde liegenden **Web of Trust** [Fei97], in dem jeder durch seinen Schlüssel die Authentizität anderer Schlüssel bestätigen kann. Die Schlüssel potentieller Kommunikationspartner können auf unterschiedliche Art und Weise verfügbar gemacht werden:

1. Man kann den eigenen Public Schlüssel in der Finger-Information oder auf einer WWW-Seite ablegen.
2. Man sendet den eigenen Public Schlüssel an einen der Keyserver [Keya, Keyb], welche untereinander diese Information austauschen, so dass jeder Schlüssel auf jedem Server vorhanden sein sollte. Dort kann er dann von Interessierten bezogen werden.

In einen nächsten Schritt muss dann die Authenzitität eines Schlüssels vom Benützer überprüfbar sein. Üblicherweise werden für diesen Zweck die angehängten Zertifikate rekursiv verfolgt. Dazu existieren zur Zeit zwei Möglichkeiten:

1. Die Überprüfung erfolgt beim Einbinden des Schlüssels in die eigene PGP-Schlüsselsammlung (“Keyring”); allfällige Verbindungsschlüssel, d.h. Schlüssel von anderen vertrauenswürdigen Benützern, müssen manuell gesucht und in den Keyring eingefügt werden.
2. Ein dedizierter, vertrauenswürdiger Server kann mit der Anfrage betraut werden [RS97, RS]. Wenn diesem Server nicht vertraut wird, kann man sein Ergebnis nachprüfen. So sind aber nur fälschlich angegebene Vertrauenspfade erkennbar, ausgelassene Pfade können so nicht erkannt werden.

Das internationale Netz von Keyservern kennt jedoch diverse Schwächen: 1. Die Anzahl der Schlüssel wächst so schnell, dass die dem Keyserver zugrunde liegende Datenbanksoftware ihre Leistungsgrenzen erreicht. 2. Die Verteilung der Schlüssel unter den Keyservern funktioniert nicht immer so zuverlässig wie gewünscht. 3. Bestehende Keys und Signaturen können nicht mehr verändert werden, weil das System nur Additionen erlaubt. Einzig sogenannte Revocation Zertifikate sind möglich: Der Besitzer eines Schlüssels muss dazu einen speziell generierten PGP Key an die Keyserver senden.

Um diese Probleme anzugehen, wurden folgende Initiativen ergriffen:

Datenbankgrösse: Es wurde vorgeschlagen, die unterliegende Datenbank zu überarbeiten, und den Rest des Systems beizubehalten; bis jetzt hat sich aber noch niemand bereit erklärt, diese Arbeit zu übernehmen.

Verteilung: Es ist geplant, die Schlüssel der Benutzer in das bestehende Domain Name System (DNS) [Moc87] einzubinden [Gil97, EK97]. Die durchgängige Realisierung dieser längerfristig skalierbaren Lösung wird jedoch mehrere Jahre in Anspruch nehmen. Auch danach wird es noch DNS-Domains geben, die keine PGP-Schlüssel unterstützen werden sowie Schlüssel, die nicht in das definierte DNS-Namensschema passen.

Authentisierte Keys: Müsste sich derjenige, welcher einen Schlüssel an einen Keyserver schickt, gegenüber diesem authentisieren, bestände die Möglichkeit diese Keys auch nachträglich wieder auf Wunsch des Besitzers von den Keyservern zu löschen. Dieses Verfahren wurde auch die Möglichkeit geben, Signaturen von Schlüsseln zu entfernen.

Synchronisation: Als Ersatz für die aktuell via E-Mail erfolgende Synchronisation wurde zuerst versucht einen zuverlässiger Dienst unter Ausnutzung von IP Mulitcast [FJM$^+$95, PSB$^+$95] zu erarbeiten [Bau98]. Ein neuer Ansatz ist der Inhalt dieser Semesterarbeit

2 Aufgaben

Im Rahmen dieser Semesterarbeit soll die bestehende Keyserver Software konzeptionell so erweitert werden, dass die Synchronistaion der Schlüssel über ein einfaches und effizientes Protokoll abgewickelt werden kann.

. . . To be filled in

Die Aufgaben, die in dieser Semesterarbeit erfüllt werden sollen umfassen:

1. eine Untersuchung des aktuellen Systemes,
2. eine Analyse möglicher Lösungen, u.a. die Aspekte Machbarkeit, Effizienz und Integrierbarkeit in die bestehende Keyserver Software,
3. den Entscheid für die beste Lösung,
4. die Erstellung des Designs dieser Lösung,
5. die Umsetzung dieses Designs in Software,
6. sowie Tests der realisierten Implementation und eine Performance Evaluation.

3 Vorgehen

- Machen Sie sich mit der Umgebung und speziell der Dokumentation und vorhandenen Software zu den Themen PGP, Keyservern und Authentisierungsverfahren.
- Evaluieren Sie die unterschiedlichen Möglichkeiten zur Lösung der gestellten Anforderungen.
- Treffen Sie die Entscheidung für eine der Möglichkeiten und begründen Sie sie.
- Erstellen Sie ein Design der von Ihnen gewählten Lösung.

- Implementieren und testen Sie diese Lösung.
- Auf eine klare und ausführliche Dokumentation wird besonderen Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte und untersuchten Varianten vor dem definitiven Variantenentscheid ausführlich schriftlich festzuhalten.

4 Organisation der Arbeit

- Mit der Betreuerin sind in der Regel wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen soll der Student mündlich über den Fortgang der Arbeit und die Einhaltung des Zeitplanes berichten und anstehende Probleme diskutieren.
- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit der Betreuerin abzustimmen.
- Nach der Hälfte der Arbeitsdauer soll ein kurzer mündlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt. Dieser Zwischenbericht besteht aus einer viertelstündigen, mündlichen Darlegung der bisherigen Schritte und des weiteren Vorgehens gegenüber Professor Plattner.
- Am Ende der Arbeit ist eine Präsentation von 15 Minuten im Fachgruppen- oder Institutsrahmen fällig. Anschliessend an die Schlusspräsentation soll die Arbeit Interessierten praktisch vorgeführt werden.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Die Dokumentation ist entweder mittels des Satzsystemes LaTeX oder mit dem Textverarbeitungsprogramm FrameMaker zu erstellen.
- Es ist ein Schlussbericht über die geleisteten Arbeiten abzuliefern (2 Exemplare). Dieser Bericht ist in Deutsch oder Englisch zu halten und beinhaltet sowohl eine deutsche wie auch eine englische Zusammenfassung, die Aufgabenstellung und den Zeitplan. Der Bericht besteht aus einer Einleitung, einer Analyse über verwandte und verwendete Arbeiten, sowie einer vollständige Dokumentation der Programme und Tools, die für weitere, darauf aufbauende Arbeiten praktisch brauchbar ist.
- Die Arbeit wird als CDrom archiviert werden. Der Student kann die vorhandene Infrstruktur ausnützen um seine Arbeit auf CDrom zu brennen.

Literatur

[Bau98] Michael Baumer. Verteilte, mittels Multicast synchronisierte Server für PGP-Schlüssel. Semester Thesis Summer Term, 1998.

[CDFT98] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP message format. Internet RFC 2440, November 1998.

[EK97] Donald Eastlake, 3rd and C. Kaufman. Domain name system security extensions. Internet RFC 2065, January 1997.

[Fei97] Patrick Feisthammel. Explanation of the Web of Trust of PGP. `http://www.rubin.ch/pgp/weboftrust.en.html`, 1997.

[FJM+95] Sally Floyd, Van Jacobson, Steve McCanne, Lixia Zhang, and Ching-Gung Liu. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of ACM SIGCOMM '95*, pages 342–356, September 1995.

[Gil97] John Gilmore. Domain name system security. http://www.toad.com/dnssec/, 1997.

[Keya] PGP keyserver network. http://www.pgp.net/pgpnet/.

[Keyb] PGP keyserver network software. http://www.mit.edu/people/marc/pks/pks.html.

[Moc87] P. Mockapetris. Domain names — concepts and facilities. Internet RFC 1034, November 1987.

[PSB+95] S. Paul, K. Sabnani, R. Buskens, S. Muhammad, J. Lin, and S. Bhattacharyya. RMTP: A reliable multicast transport protocol for high-speed networks. In *Proceedings of the Tenth Annual IEEE Workshop on Computer Communications*, September 1995.

[RS] Michael K. Reiter and Stuart G. Stubblebine. AT&T pathserver. http://www.research.att.com/~reiter/PathServer/.

[RS97] Michael K. Reiter and Stuart G. Stubblebine. Path independence for authentication in large-scale systems. In *Proceedings of the 4th ACM Conference on Computer and Communications security*, pages 57–66, April 1997.

[Sta94] W. Stallings. Pretty Good Privacy. *ConneXions*, 8(12):2–11, December 1994.

Zürich, den 30. September 2002