Design and Implementation of Secure Trusted Overlay Networks

Matthias Jacob

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

Adviser: Edward W. Felten

September 2009

This thesis has been generated using the ITC Stone^{TM} font.

To my parents

Abstract

Denial-of-service attacks, malicious routing updates, and online identity theft are clearly on the rise on the Internet, costing the US industry billions of dollars. In reaction, there is a large effort to design new technologies such as Trusted Computing that solve many of these problems efficiently. However, state-of-the-art systems for anonymous communication have various weaknesses against traffic analysis and are often designed for one specific purpose. So far, Trusted Computing has not been considered for improving the efficiency of Internet anonymity and privacy and building a general-purpose architecture to solve the problem.

In this thesis we describe the design and implementation of Secure Trusted Overlay Networks (STONe). STONe is the first system for general-purpose anonymous communication that is entirely based on Trusted Computing. STONe significantly improves anonymous communication on the Internet and makes three main contributions. First, STONe uses Trusted Computing to protect against Byzantine Failures on the network stack to provide an overlay network for scalable, efficient secure routing, and end-to-end communication. This prevents many active denial-of-service attacks on an anonymity network and provides a foundation for more robust protection against traffic analysis. Second, STONe is the first system to provide anonymous routing through load-balancing by random routing previously used for local cluster networks. This turns out to better protect against most existing traffic analysis attacks. Such attacks have yet been difficult to come by, namely the Predecessor Attack and the Intersection Attack. Third, on the application-level, STONe provides application-level anonymity through trusted anonymous sockets and a trusted name service, an inexpensive trusted certification mechanism with one-way per-session authentication. We implemented and evaluated a prototype of STONe on PlanetLab and show that it significantly outperforms state-of-the-art systems for anonymous communication at the expense of additional Trusted Computing hardware.

Acknowledgements

I was very glad to have Ed Felten as my advisor. He gave me the freedom to work on anything I enjoyed and helped me finally succeed in my research. Dan Boneh, Vivek Pai, Kai Li, and Jennifer Rexford as my thesis committee members provided me with valuable feedback and guidance in laying out the research and writing my thesis. In addition, I would like to thank especially Dina Katabi, and also Mariusz Jakubowski and Ramarathnam Venkatesan for successful collaborations during my graduate school career. And of course, I am grateful to have had many nice friends at different universities and research institutions I have had the chance to work at. It has always been a good time.

Table of Contents

Chapter 1

Introduction

Privacy is becoming increasingly important on the Internet. Sophisticated surveillance tools that can reconstruct anyone's HTTP and EMail traffic are now commercially available. These tools are becoming extremely powerful, even causing the FBI to deploy them as a replacement for their in-house surveillance system, Carnivore [6, 13, 145]. Consequently, Internet users increasingly need systems that ensure the *privacy* of user identity and communication, unless the user voluntarily discloses this information.

Even though encryption keeps the content of the messages secret, IP addresses not only provide routing information but unfortunately also reveal the identity of users. Specific traffic properties such as inter-packet timing give an adversary further clues about the communicating parties and the type of traffic. Therefore, a network that provides private communication requires transparent protection against these types of *traffic analysis attacks*. Traffic analysis is a long-standing and hard problem.

In addition to anonymous routing, end-to-end anonymity is a neglected problem as well. Name server requests are a vital information source for a privacy-intruding adversary. Credentials in the system are rarely anonymous, and it often becomes easy to fake identities. Application endpoints like web servers are often able to distinguish between messages they receive from inside and outside the anonymity network. The receiver may not have an incentive to accept messages from inside the anonymity network and may just drop these messages, thereby forcing the sender to reveal its identity. These are all significant shortcomings, and an anonymity system with integrated end-to-end support has stronger security and anonymity properties than a proxy network that has a peer-to-peer-based system such as Gnutella running on top of it. Gnutella is known to have privacy problems [43].

In this thesis we present STONe, Secure Trusted Overlay Networks, to address the aforementioned problems. STONe *decouples message forwarding from traffic analysis protection* and *integrates end-to-end anonymity with anonymous routing*. It achieves significantly better leverage on performance, resilience, and anonymity than previous systems for anonymous communication. But as a trade-off STONe requires a Trusted Computing infrastructure which is only available in new CPUs [99, 28].

1.1 Anonymous Communication

The Internet by itself does not provide any protection for anonymity. Every IP packet clearly reveals the source and destination address of the endpoints, and, unfortunately, the routers require this information to ensure optimal routing in the network. An adversary with access to the network or routers is able to carry out traffic analysis attacks, and in addition the endpoints are able to see the peer's identity.

So far there exist a variety of systems for anonymous communication, from sending anonymous email [58] to anonymous web browsing [131, 204]. But building an efficient system for general-purpose anonymous communication that is robust against a wide range of attacks becomes a very challenging task. Often these state-of-the-art techniques are based on a variation of intermediate proxies or broadcast techniques. They either do not provide full protection against traffic analysis [131, 204], have high latencies [62], are limited to small-scale networks [148, 59], or are not resilient against failures [58]. In addition, they are prone to some dangerous traffic analysis attacks, most importantly the predecessor attack [198, 171], in which an adversary analyzes packet header information to find the sender. They are also vulnerable to intersection attacks [131], in which the adversary measures traffic properties like volume and timing to find a subset of nodes that comprise the possible communication path. The dilemma is that protocols protecting against the latter attack use random walks to blur the path between the sender and the receiver and hide the associating IP addresses. However, when the route is changing frequently an adversary needs to compromise considerably fewer nodes to catch the desired packet header information. To our knowledge there is no satisfactory solution that solves this problem efficiently.

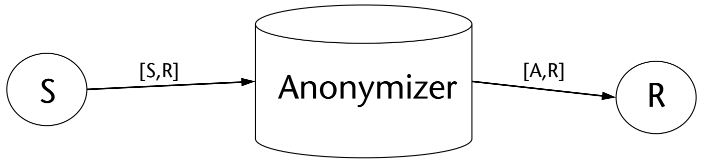

Figure 1.1: Single proxy network for anonymous communication. The sender forwards messages to the Anonymizer proxy, which then propagates them to the receiver, replacing the original sender address with its own address.

The simplest solution for providing Internet anonymity is to use a trusted proxy such as Anonymizer [2]. Figure 1.1 shows the scenario: The sender forwards its messages to the proxy, and the proxy then propagates the messages to the receiver.

The receiver only sees the proxy's IP address and returns messages to the sender using that address.

However, being a single point of failure, the proxy has to be fully trusted, similar to a certification authority that certifies public keys for Internet identities. Whenever an anonymity proxy leaks information about a forwarded packet it compromises the whole system. In addition, trusted backup servers have to be ready in case of a failure to avoid disruption of the anonymity service in case of a failure or overload situation. The proxy hides the IP addresses, but end-to-end timing still depends on the round-trip time of the individual connection and is not random, thus, giving clues to an adversary about the IP addresses.

This is particularly dangerous when a government wants to seize communication data for tracking a dissident. With a single proxy it is fairly easy to do, because the information may be stored in a central database.

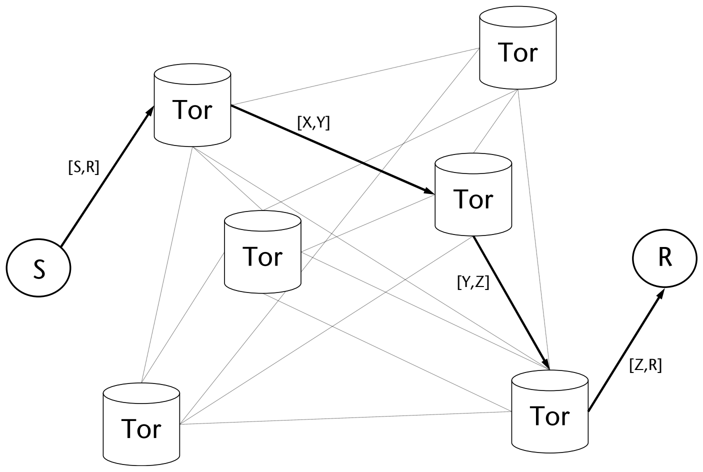

Figure 1.2: Distributed proxy network for anonymous communication. The sender forwards messages to an entry node in the Tor network. The sender's entry node propagates the message to an exit node, which then sends the message to the receiver. Tor resets the path after some time interval. The entry and exit nodes know who the sender and the receiver is.

Instead of a single proxy that has these shortcomings, distributed proxies that eliminate the single point of trust can be used for anonymous communication. In the virtual world systems like Tor [75], as shown in Figure 1.2, have become the state-of-the art technology for anonymous communication using this scheme. The sender picks a random set of proxies and uses them for message forwarding. The sender encrypts the stacked IP headers in layers, and every proxy strips off layer after layer, such that every proxy can only see the previous and next hop. The endpoints do not have to trust the proxies anymore and all messages go through an arbitrary set of untrusted nodes. When a node fails, the sender picks a different set of proxies for message forwarding.

A distributed proxy that contains untrusted nodes is prone to traffic analysis attacks – the *predecessor attack* significantly degrades almost every anonymity protocol in distributed environments when paths are changing frequently [198]. The predecessor attack exploits the fact that every node knows its predecessor on the Internet, but the sender and receiver never change during a communication session. When a path changes frequently an adversary can detect whether its predecessor is the sender or not. However, when the path remains static for too long the system is prone to *intersection* and *timing attacks* [131, 41]. End-to-end timing as well as the load on every node tells an adversary where packets traverse the system, and it is easily possible to reconstruct the sender and receiver's IP address on the communication path [131]. Unfortunately, one of the two attack methods always seems to be applicable, so that it is very difficult to achieve good protection against traffic analysis.

Similar to the single proxy, most distributed proxies used for anonymous communication have a scalability problem. A sender has to discover all proxies in the network and learn the public keys to construct the anonymous message before sending it off to the first proxy. Doing this discovery efficiently would require some underlying structure that reduces the number of key exchanges. Without encryption the message's receiver is revealed [148]. Furthermore, without admission control the senders are responsible for route selection and need to play fair, otherwise they could congest a proxy and slow down performance of the distributed proxy network.

A general problem that affects single and distributed proxies on the Internet is the limited 32-bit IP address space. An adversary can always try to guess the correct node, and she is always right with probability $p = 2^{-32}$. For example, if an adversary wants to find out where a data stream is sent from, she can attack arbitrary nodes with Denial of Service attacks and then wait until the stream becomes weaker. Because of the number of legacy routers and applications in the Internet a global move to IPv6 with a 128-bit address space in the near future seems unlikely. On the other hand, NATs do not really solve this problem, because they generate a non-uniform address space in which many nodes have the same network address and can therefore not be used as proxies for anonymization.

Summarized, there are two goals in anonymous communication systems:

Anonymous Identities and Credentials When parties communicate on the Internet it is their goal to hide their identities, such as IP address or DNS name, from an external adversary as well as from other parties participating in the communication. But they still want to be able to verify some credentials to a peer node.

Protection against Traffic Analysis The parties want to protect their asynchronous Internet traffic against an external eavesdropper whose goal it is to analyze traffic to determine which parties are participating in communication.

Trusted Computing already provides support for security in distributed systems [87], and STONe relies on the same three features of Trusted Computing in its anonymity protocol – Strong Process Isolation, Remote Attestation, and Sealed Storage – to enhance privacy in network communication. First, *remote attestation* [51, 21] allows nodes to anonymously authenticate themselves to their peers, establish trust, and form a *trusted overlay network*. Second, *strong process isolation* shields memory from spyware and attackers on the same host by isolating memory pages [133, 99, 28]. Third, *sealed storage* provides secure storage for keys and ensures safety of remote attestation [21].

Our Secure Trusted Overlay Network (STONe) consists of multiple building blocks:
(1) Efficient Protection Against Traffic Analysis: STONe protects against traffic analysis using *self-mixing* by applying random routing to a regular network topology such as a hypercube. This ensures uniform traffic patterns with minimal network congestion. This design provides mixing of network packets without explicitly using a high-latency mix network that is not useful for low-latency anonymous communication [187]. STONe further quickly isolates compromised nodes to minimize the impact of the predecessor attack by *using Trusted Computing to detect Byzantine failures* such as software bugs.
(2) Scalable and Robust Anonymous Routing: STONe improves scalability of anonymous routing because it *encrypts packet headers hop-by-hop* instead of using a circuit-based approach like in onion-routing [75]. The *structured overlay network in STONe* ensures that nodes can enter and leave the network quickly without interrupting communication of other nodes, thus eliminating single points of failure and congested network nodes. In addition, STONe also *protects the network against active attacks*, such as Denial-of-Service from external nodes, which would harm service and thus anonymity at a given time.
(3) Anonymous Sockets and Name Servers: STONe implements *anonymous TCP and UDP socket endpoints* for an application. Only truly random IP addresses are visible outside STONe. Such an anonymous IP address hides identity and location and is different for every new session the socket uses. In STONe, anonymous IP addresses are not only network addresses but also anonymous authenticators to ensure that the anonymous IP address is indeed correct. Further, STONe contains TNS, the *Trusted Name Service that maintains and verifies self-certifying anonymous credentials*. It anonymizes name server queries and maps names to anonymous pseudonyms that certify themselves with self-containing public keys.

Our evaluation shows that STONe's performance impact is much less compared to existing anonymity systems that do not use Trusted Computing. STONe's throughput approximates expected TCP throughput and exceeds state-of-the-art system Tor by 30% [75]. Also, our experiments verify that STONe scales up to a significant number of nodes on PlanetLab with random arrivals and departures, while maintaining routing stability and low overhead. Thus, STONe can optimize anonymity for locality and avoid congestion situations, and as a result its average latency is only about half of Tor's latency. Finally, our results confirm the expected benefit of random routing: it significantly improves the system's robustness against traffic analysis by maintaining scalability and resilience at the same time. We summarize our contributions as follows:

- We have designed, implemented, and evaluated STONe, a distributed infrastructure for certified and anonymous communication that is robust against substantially more traffic analysis attacks, more resilient, and more efficient than previous systems for anonymous communication such as Onion Routing, Mix Networks, or Crowds [185, 58, 148].

- We are the first to apply random routing over a regular network topology on the heterogeneous Internet to achieve load balancing and self-mixing of network packets, and thus anonymous communication without explicit mixes that hamper low-latency communication. Previous approaches use less secure random walks instead [185, 58, 148].

- STONe is the first system that combines these three important properties for anonymous communication: Resistance against the Predecessor Attack [148], uniform traffic patterns and indistinguishable communication paths [131], and scalability [59]. Further, it provides protection for traffic anonymity to disguise the type of content, e.g. media stream vs. email traffic. In particular, to provide the latter it is usually necessary to send expensive cover traffic to disguise the type of traffic [121].

- Further, we demonstrate that STONe is indeed a general-purpose system designed for both low-latency and high-throughput communication: We build two applications, *Anonymous Instant Messenger* and *Anonymous File System*. Both applications have privacy issues that are hard to solve. Using a PlanetLab implementation, we demonstrate that our system achieves reasonable performance while preserving privacy.

Chapter 2

Trusted Overlay Networks

Byzantine failures are one of the most general problems in distributed systems. For anonymous communication they pose a particularly significant threat that requires protocol designers to downsize performance and scalability to work around these problems. Compared to a *fail-stop* failure that causes a machine to crash, Byzantine failures like software bugs cause a machine's behavior to become unpredictable. If the Byzantine failure is even an intended malicious attack, an adversary takes control over the whole machine and uses it to launch further attacks. This threat is real – adversaries can easily get access to tens of thousands of compromised computers in so-called BotNets and use them to launch DDoS attacks on web servers or networks [103]. When the adversary controls the machine, she is also able to monitor all communication channels traversing through that machine, and therefore, BotNets are also a threat to anonymity.

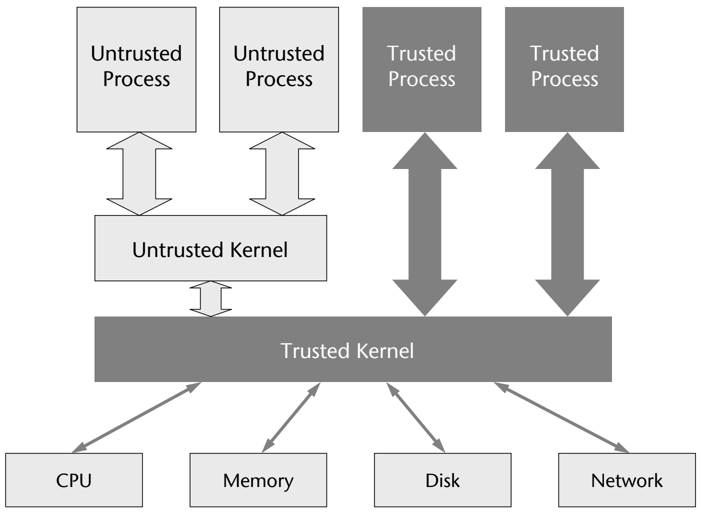

Figure 2.1: **Trusted Computing:** Trusted versus untrusted processes on a Trusted Computing node

Trusted Computing, as proposed by several manufacturers [99, 28, 21, 79], provides improved proactive protection against Byzantine failures on a local platform using two distinctive primitives: *Strong Process Isolation* and *Remote Attestation*. In addition to virtual memory *Strong Process Isolation* protects trusted processes against attacks from a compromised OS through virtualization [87], and thus isolates Byzantine failures from the rest of the machine that would otherwise make the platform vulnerable. A process becomes trusted, and thus part of the *Trusted Computing Base (TCB)* only when it completes attestation locally. To complete attestation the process verifies to the TCB by signing a nonce that all software from the application to the BIOS is trusted, i.e. the nonce is on the list of trusted software. Otherwise the process remains untrusted. This separation is shown figure 2.1.

Remote attestation is an application-based mutual protocol between two TCBs that verifies to the peer TCB that the platform is trusted. Similar to local attestation on the TCB itself, it verifies to the peer node that all software from the BIOS to the application is trusted. Remote Attestation is built into the TCB and can be implemented in various fashions [165, 51, 114, 87]. It either exploits hardware properties such as hardware-specific clock skew or relies on cryptographic primitives such as group signatures that rely on keys built into hardware. Group signatures are a signature scheme that preserves identity from others. The signing node signs a nonce n_b of the binary using its private signing key, and every other node in the group is then able to use the global verification key to verify the group signature. Every TCB has a list of valid nonces n_b that are trusted, and without forging or breaking the hardware it is not possible to get the system to succeed in remote attestation. This significantly raises the bar for adversaries to compromise a node – for example, it would not be possible to compromise a set of nodes by spreading a worm across the network.

A group of interconnected TCBs forms a *Trusted Overlay Network*, the platform for STONe. The Trusted Overlay Network isolates processes from Byzantine Failures and reduces the likelihood of security bugs, because software is trusted. In such a closed distributed system a software-based compromise affects either *all* nodes or *none* of the nodes, which is similar to *fail-stop* behavior.

In addition to the benefits in robustness against Byzantine failures, unfortunately, there are also some controversial issues in Trusted Computing that need to be discussed. This is mainly because it enforces policies and lets an outside server take control:

(i) It is hard for a user to verify that Trusted Computing has been *implemented correctly* and does not leak any information through hidden backdoors.

(ii) Trusted Computing can pursue *anti-competitive behavior* by implementing restrictive policies in TCBs that lock out certain software from the platform under the premise that the software is insecure.

(iii) *Systems maintenance* becomes hard when operating systems have to be updated frequently, since on every update old nonces for attestation have to be removed from the list of trusted software.

(iv) A local administrator can be *restricted* because Trusted Computing adds another privilege ring around the operating system. Only a global remote administrator who might not even be known to the owner of the platform could have all administrative permissions.

(v) When an adversary *compromises a Trusted Computing system*, it is usually a total break-in, and the adversary is able to learn everything including the TCB's secret key used for attestation. However, the probability that such a break-in occurs is assumed to be very small.

There are several mechanisms in place to overcome these problems. Regarding genuity of the Trusted Computing implementation the hardware manufacturer itself has to be trusted. The manufacturer's damage from bad publicity needs to outweigh the benefit from the backdoor.

Owner override addresses some of the issues related to access rights [162]. In owner override the owner can modify configurations or even remote attestation as long as it is proven that the owner is making the changes and not a virus or malicious application. The problem with owner override is that it undermines the security of Trusted Computing and allows cheating in online games, illegal copying of protected content, etc.

To make the attestation process more transparent the TCB can use techniques such as semantic attestation [93]. In semantic attestation the signature is not computed by a hash function but by applying a function that captures the properties of the program like in proof-carrying code [134]. In this case no additional list of trusted software needs to be distributed, since attestation only checks the program for given functionality and not what kind of software it is.

When an adversary breaks into the Trusted Computing hardware she is able to learn almost everything on the compromised machine. Deniable cryptography against the "rubberhose attack" could potentially shield such compromises but has been fairly inefficient to implement [125, 53]. In deniable cryptography a plaintext P is encrypted such that the corresponding ciphertext C decrypts to P under key k_1, and under key k_2 C decrypts to a different meaningful plaintext different from P. When an adversary knows key k_2 she thinks that she has found the correct key.

2.1 Distributed Applications on Trusted Overlay Networks

Trusted Overlay Networks are the base for STONe. Numerous distributed applications can benefit from Trusted Overlay Networks that would otherwise have to cope with Byzantine failures. Algorithms that protect against Byzantine failures are often costly, since they require replication [57, 19]. Trusted Overlay Networks provide a flexible infrastructure that strengthens any distributed systems. Here we give a few examples of distributed applications that potentially benefit from the Trusted Overlay Network architecture:

Ad-Hoc Networks
: Ad-Hoc networks are untrusted because the owner of a node on the ad-hoc network has access to all submitted data and can eavesdrop on data or manipulate and drop data. TON protects transmitted data against tampering and traffic analysis. TON also eliminates the problem of free-riders who use the network infrastructure without forwarding any data.

Distributed File Systems
: As mentioned in the last section, any distributed peer-to-peer file sharing application or distributed filesystem has benefits on TON, because an adversary would not be able to compromise or inject nodes with malicious content. Normally, distributed file systems have to implement replication in order to protect against maliciously modified files [57].

Global Computing
: Global Computing such as SETI@Home is another application for Trusted Overlay Networks [19]. When an adversary tampers with the particular result of some nodes the global result is bogus. To protect against this attack computation has to be replicated, which is expensive. In Trusted Overlay Networks replication is unnecessary, since the nodes are protected by Trusted Computing hardware.

Electronic Voting or Consensus Systems
: Any system for consensus or in particular electronic voting has to be robust against Byzantine failures. In particular, the policies have to be enforced, such that entities do not cast a vote twice or jam the consensus process, as, for example, in Dining Cryptographers.

Instant Messaging
: Instant Messaging is a distributed system that relies on a trusted central server for directory lookup and message forwarding. This one single trusted entity can fail or get compromised. With Trusted Overlay Networks we can distribute the functionality of this one single trusted entity across the whole network.

Chapter 3

Model and Definitions

Before we get into the design of STONe in this chapter we explain the underlying models for security and networks used in this thesis as well as potential attacks and attack goals considered in STONe's design. A reader who is only interested in the systems design and implementation can skip this chapter and go directly to chapter 4.

3.1 Communication Model

A *network* is a set of *nodes* and *links* that maps to a graph $G = (V, E)$ of vertices and edges. Nodes are connected with bidirectional links and communicate by sending data over these links. We distinguish between *synchronous* and *asynchronous* communication. Synchronous communication always depends on a global clock, whereas nodes are allowed to send data at any time in asynchronous communication. We always assume that nodes are able to separate real data from noise in asynchronous communication.

Our model assumes asynchronous Internet communication that uses a standard TCP/IP stack, as in commodity operating systems. Links are bidirectional point-to-point connections, and messages are forwarded hop-by-hop. Routing table information gets updated using some standard link-state protocol like OSPF. Single routers in the network are untrusted and can be administered by different authorities.

An application sends traffic at any time through a *communication channel*, whenever data is available from the user. This channel can either be connection-oriented or connection-less. A forwarding node buffers received packets when a potentially high system load does not allow forwarding more packets. Otherwise the node immediately sends a packet off to the next hop. We assume that communication channels last over an extended time period and that participating nodes repeatedly exchange messages over the same communication channel. Every communication channel has a path through the network. This path is not necessarily static and may change over time. In the beginning this path gets initialized, but the network may do multiple *path* *reformations* during the duration of the communication channel for multiple purposes.

Users initiate *sessions* to exchange traffic with other nodes in the network, and they do this arbitrarily. A session depends on some application and can for example consist of web browsing, peer-to-peer communication or instant messaging. Studies have shown that session arrival is best modeled according to a Poisson distribution, but packet arrival times are usually distributed according to heavy-tailed distributions [141].

Especially user sessions are important for anonymity, and we need to distinguish between *interactive sessions* and *non-interactive sessions*. In an interactive session two nodes send request and reply messages back and forth. The requesting user waits for the answer or retransmits the request before she sends out the next message. Interactive sessions can have distinctive patterns that give an adversary extra information. For example, when a user opens a browser, and the browser always points to the same user-specific homepage, it gives an attacker some extra information. Identifying a web page – given the number and lengths of encrypted packets – is not hard [183]. In contrast, a non-interactive session always consists of a steady unidirectional stream of messages. We use the term *packet* for session-layer data and the term *message* for network-layer data. For example, an HTTP request would be a message and IP data would be a packet. Data units in STONe's network layer are called a *fragments*.

A communication network is characterized by its *diameter* and its *bisection width*. The diameter defines the maximum distance between any pair of processes, and the bisection width the minimum number of edges that have to be removed in order to disconnect the network into two halves with identical number of processors [120]. The diameter defines the maximum latency in the network. The bisection width is a critical performance factor in a network, since it describes the network bottleneck under congestion. A ring with n nodes, for example, has a relatively poor performance, since its bisection width is 2 and the diameter is $\frac{n}{2}$. In contrast, a 2-dimensional mesh with n nodes has a bisection width of $\sqrt{n}$ and a diameter of $2\sqrt{n}$, whereas the maintenance cost per node is almost the same. Furthermore, a *matching* of a graph or network is a set of edges, such that no two of them have a vertex in common. The largest possible matching on a graph is a set of $\frac{N}{2}$ nodes, and this is called a *perfect matching*. A graph with a perfect matching has a bisection width of $\frac{N}{2}$.

The *congestion* of a link is the expected queue length of messages over this link at any given time. A congestion of 1 means that there is no congestion, and the network can always work efficiently. When a link has m messages queued up the m-th message has to wait for (m-1) steps until it gets forwarded.

3.2 Systems Model

In this work we assume an asynchronous distributed system model. The network consists of N nodes that are fully connected through the Internet. Every node in the distributed system can have a different administrator and also run different versions of the application.

Applications communicate whenever data is available to send. Asynchronous distributed systems have communication uncertainty, because nodes may crash and remain undetected. In addition, our model allows Byzantine failures, such that an adversary can compromise nodes and tamper with communication channels.

Protection against Byzantine failures is achieved by sandboxing and black-boxing from Trusted Computing hardware. Chapter 2 contains explanations about Trusted Computing and Trusted Overlay Networks.

3.3 Security Model

In our security model we tolerate an adversary that can launch any type of software-based attack. We model the adversary as a Dolev-Yao attacker. A Dolev-Yao attacker is a non-deterministic process that has complete control over the communication network [76]. The attacker can introduce packets into the network when she has access to the untrusted operating system on a node ; these packets may have fake identities, mount a DDoS attack or try to introduce Trojans into the overlay nodes. Also, an attacker might pose as a honeypot to intercept all communication between a sender and a receiver, which is called a man-in-the-middle attack. Sybil attacks are possible, when an adversary tries to compromise the network with her own compromised nodes [56]. We distinguish between an adversary with physical access to the machine and the adversary with only virtual access. The adversary with physical access can eavesdrop on network-layer packets. The other may not be able to.

A second type of adversary is an external attacker with access to the network. This adversary can listen to network communication or tamper with traffic. Usually, this kind of adversary needs to have significant power equivalent to an Internet Service Provider (ISP).

STONe is a communication infrastructure that protects against most common network attacks. Specifically, it protects against the following attacks:

Denial-of-Service Attacks

An adversary can launch DDoS attacks on other nodes in the network, either from within the network or as an outsider. These DDoS attacks have multiple layers: First, the adversary can flood the network with packets to saturate the network bandwidth. It is typically quite hard to protect against this type of attack, so the goal is to achieve an improvement over a normal TCP/IP-based network communication channel. Second, the adversary can do protocol-level attacks like SYN floods [20]. And furthermore, it can launch application-level attacks like HTTP floods [103]. In these cases either the network infrastructure is obstructed or application-level services do not work anymore.

Routing Attacks
: Sabotaging the network as well by deliberately (i) rerouting messages, (ii) reordering messages, (iii) dropping messages, (iv) launching DDoS attacks against servers, or (v) manipulating NodeIDs is another possible attack. Standard routing protocols such as BGP [150] have these problems. For example, an adversary can tamper with the protocol that updates the routing tables and pretends link failures. The adversary thus decreases the performance of the network.

Traffic Analysis
: An adversary is able to carry out Traffic Analysis in two different ways: Because software has bugs an adversary could either *compromise the OS kernel or another process* to measure network packet data, or she could *probe nodes remotely*. When nodes are compromised it is straightforward to eavesdrop on traffic to either find specific targets in the network or to locate and track communication to violate privacy. An adversary investigates communication patterns over multiple nodes to find out which parties are communicating. On the other hand she is also able to analyze traffic characteristics to see whether, for example, some party is running peer-to-peer traffic or browsing the web. By remotely probing nodes only a limited set of attacks is possible. In this attack an adversary determines the load of the individual nodes or measures the timing for encryption [131].

3.4 Privacy Model

According to Merriam-Webster, *Privacy* is "the quality or state of being apart from company or observation". For example, any unauthorized intrusion is a breach of privacy. On the Internet, breach of privacy is often associated with stealing confidential information such as credit card numbers. In common peer-to-peer protocols, private information leaks at different places [43]. In this thesis we consider the following attacks on privacy:

Passive Logging Attack
: Logging and intersecting information on the Internet is a large threat. An adversary who logs any type of information on the Internet is considered a passive logging adversary. Logging information that is freely available is nothing illegal, but the amount of available information may not be authorized by the logged

entity, because people are unaware of technical options they have available to hide this information. On the Internet, for example, several anonymous routing systems are available to protect information about an IP address leaking to a website (e.g. [75, 148, 2]).

Phishing Attack

In a Phishing attack an adversary uses social engineering to fool somebody into a fake network site. Phishing attacks set up forged websites of real companies on which an adversary wants to obtain any confidential information such as credit card or social security numbers. The forged websites pretend to be major companies such as PayPal, Citibank, or EBay. Often, Phishing adversaries send official-looking emails to their victims that point them to their forged website. The main vulnerability Phishing adversaries exploit is people's superficial trust in companies' logos and letterheads. These scams are sometimes even so hard to distinguish that there are Phishing IQ tests [16]. Electronic certificates solve this problem, since they clearly identify the company the website belongs to [22]. Phishing does not require that many people get tricked into the scam, but the number of circulating email messages is so large that even a few hundred users are sufficient to cause a significant amount of damage [111].

Pharming Attack

In a Pharming attack an adversary compromises the Internet name server directly [111]. Whenever a client contacts the name server it gets redirected to the adversary's website. Pharming attacks require the adversary to exploit some actual technical vulnerabilities, whereas a Phishing adversary exploits human weakness.

Censorship Attack

In contrast to the previous passive attacks that try to gather information from an individual, the censorship attack is an active privacy intrusion attack. When someone publishes legitimate information and an adversary suppresses or deletes this piece of information it is also an unauthorized intrusion. On the Internet censorship attacks are generally hard because only ISPs can censor information globally by disconnecting servers. In a local environment firewalls are usually being used to shut off information from the Internet. Censorship is not only intrusion into the author's privacy, but also into the reader's privacy, because someone else decides which information people are able to obtain. Of course, there have to be methods for blocking illegal content.

Impersonation Attack

Another type of a privacy breach occurs when an adversary impersonates someone's identity after stealing significant identification (aka identity theft). This causes massive privacy intrusion because the adversary can impersonate another person online and cause serious damage. Like the censorship attack impersonation is also an active intrusion attack.

Traffic Analysis

Traffic Analysis by itself is also a breach of privacy. When an adversary analyzes traffic to uncover the identities of the sender and the receiver she violates privacy. We discuss the underlying anonymity model of Traffic Analysis in the following paragraph.

3.5 Anonymity Model

Anonymity is defined as "the state of not being identifiable within a set of subjects, the *anonymity set*" [143]. An anonymity set is therefore the set of all distinguishable subjects in the system. Anonymity helps to protect privacy but falls short of real privacy.

Someone could say that *cryptography* provides anonymity because it provides operations to randomize messages [174]. In particular, secure multiparty computation [89, 146] and secret sharing [167, 45] are related to anonymous communication. However, this is only one part of anonymity. Messages have multiple properties – for example, a network message has a certain timing behavior that cryptography cannot hide, and only synchronous communication, as in secure multiparty computation, can solve this problem. If the anonymity set is a pool of messages it can be identified based on the timing behavior. Furthermore, if the anonymity set is the set of all nodes in the network, cryptography alone does not help.

On the other hand *steganography* can solve some of these problems, but this is only partially true. Steganography hides messages without using cryptography by embedding messages, for example, in digital images or TCP protocol headers [172, 106]. However, it is hard to prove security, and often steganography can be broken. Steganography not only disguises message content but also hides the actual message transmission. Therefore, protection against traffic analysis is in some sense steganography.

In our anonymity model for communication networks we define an adversary's goals for *traffic analysis*. In a network of N nodes an adversary himself can be the sender, the receiver or a third party. This adversary controls any type of nodes within the network – senders, receivers or internal idle nodes – and pursues the following goals [148]:

Sender Anonymity

The sender of a communication channel wants to protect her anonymity against a Traffic Analysis adversary. The adversary could either be the receiver or any limited set of nodes in the network.

Receiver Anonymity
: Similarly, the adversary's incentive is to find out the correct receiver. Any node along the path from the sender to the receiver has to know how to forward the packet, but there exist techniques that provide this message forwarding without revealing the receiver's identity.

Unlinkability of Sender and Receiver or internal nodes
: When the adversary already knows the possible candidates for senders and receivers of a particular communication channel in the network, her goal is to link them together. She may also know the sender or receiver already but wants to uncover the other party.

Locality of Nodes
: In some cases the adversary wants to find out where a certain node is located, either in its logical position in the network defined by its neighbors, or in its absolute geographical position. This attack goal is independent from the previous ones where the adversary's only goal is to detect senders and receivers.

Traffic Characteristics
: In addition to the sender and receiver identity of a communication channel the actual traffic characteristics provide important information to the adversary. She can then explicitly state that some node downloaded files from certain sites, and she may even be able to reconstruct the content.

Activity Monitoring
: In this case the adversary wants to determine which node is active or online at a given time. This is also some form of anonymity and furthermore helps to break unlinkability, since nodes have to be active when they participate in a communication channel.

Furthermore, an *adversary with external resources* has an advantage against an adversary without external resources. The adversary with external resources can communicate with compromised nodes through her own communication network. She can use the network to reroute messages and has additional computational power to analyze traffic logs she collects.

We also distinguish between *internal and external adversaries*. An external adversary can only observe traffic and send packets on the links, but an internal adversary has full control over the nodes.

Specifically, in STONe we are considering the following popular traffic analysis attacks that are weaknesses of existing systems for anonymous communication:

- *Predecessor Attack*
  The predecessor attack is a common way to compromise sender/receiver ano-

nymity. In this attack an adversary exploits the fact that many systems against traffic analysis use frequent path reformations to simulate a random walk over a graph (e.g. [75, 148]). However, sender and receiver never change and are therefore clearly distinguishable from other nodes. When an adversary observes predecessors of network packets over time on a limited set of nodes, and the network packets always come from the same node, an adversary can conclude that this node must be a sender. This attack only works when the adversary is able to clearly identify the end-to-end connection related to the current message as well as its predecessor [198, 171].

- *Intersection Attack*
  In an intersection attack an adversary monitors properties at nodes and correlates the collected information. For example, in a partially connected network, when the adversary discovers the sender's neighbors in the network, she is able to reduce the sets of nodes that belong to the communication path. Or, in an unsynchronized network an adversary can analyze traffic volumes to correlate possible communication endpoints [67]. Alternatively, an adversary might match the length and content of messages along different links and use this information to reconstruct the communication path [34, 138]. This information can also be used to confirm some hypotheses about the traffic pattern [68].

- *Passive Logging Attack*
  Network sites are logging network addresses of clients that have accessed the system for maintenance reasons, and an adversary can abuse this information and collect an access log of users connecting to the site, such as a web server access log. A public Internet client by itself cannot exchange or hide its network address [197].

- *Timing Analysis*
  In a Timing Analysis attack an adversary measures the message inter-arrival times on the set of nodes she controls. She then correlates the measured information from different nodes and when the information matches she concludes that they must have forwarded the same messages with high probability. This requires that the adversary is able to distinguish and identify messages on the network [121].

- *Membership List Attack*
  By collecting information about the time when nodes enter and leave the network, an adversary is able to narrow down the anonymity set from the set of all nodes in the network [9, 41].

Anonymity Measure The common measure for anonymity is the *entropy of the anonymity set*. Entropy is a measure for randomness that has the following three assumptions [168]:

- A small change in the probability of membership $p(i)$ in the anonymity set should only change the entropy by a small amount.
- When all occurrences i are equally likely then increasing one set of occurrences always increases entropy.
- The entropy of two sets of occurrences is the weighted sum of the entropies of the two sets.

The entropy function is then defined by

$$H(x) = -\sum_{i=1}^{n} p(i) \log_2 p(i).$$

In our anonymity model we require randomness of the anonymity set. It should be hard for an adversary to reduce the anonymity set to the nodes he is interested in. Then, as a logical consequence, anonymity is the entropy $H(\mathcal{A})$ of the anonymity set $\mathcal{A}$ divided by the maximum entropy H_M [163, 72]:

$$d(\mathcal{A}) = \frac{H(\mathcal{A})}{H_M} = \frac{-\sum_{i=1}^{N} p_i \log_2(p_i)}{\log_2(N)}$$

d is also called the *degree of anonymity* [72]. The degree of anonymity describes the amount of information about the anonymity set the system is leaking. When $d \to 1$ all nodes appear to be a solution to the anonymity attack. If, however, $d \to 0$ the attacker is successful and can isolate an element from the anonymity set. This definition of anonymity as the entropy follows the randomness measure of other disciplines, such as cryptography and steganography.

3.6 Notations

3.6.1 General Notations

This thesis uses most standard conventions for mathematical notations. When we use $\log$ it always means $\log_2$ unless it has an explicit base b as in $\log_b$. $\ln$ refers to the natural logarithm with base e.

Table 3.1 shows the standard terms used in this thesis. These terms describe keys and identifiers in Trusted Computing or STONe, or describe some properties of the network.

Symbol	Description
N_{max}	Maximum number of nodes in STONe; this value depends only on the length of STONe addresses (n= $\log N_{max}$= 64)
r_j	j-th n-bit random number in the internal pseudo-random number generator of Trusted Computing
N_{STONe}	Number of nodes in STONe
S_{STONe}	128-bit secret shared by all STONe nodes
$h_{TC}(\cdot)$	Hash function of the Trusted Computing platform
$h_{STONe}(\cdot)$	Hash function of STONe; $h_{STONe}(m) = h_{TC}(S_{STONe} \mid m)$
PK_{TC_i}	1024-bit RSA public key built into the Trusted Computing platform of node i
SK_{TC_i}	1024-bit RSA secret key built into the Trusted Computing platform of node i
K_{TC_i}	128-bit AES secret key built into the Trusted Computing platform of node i
$K^j_{STONe_i}$	128-bit secret key for stream cipher between STONe nodes i and j
$DH^t_{STONe_i}$	Diffie-Hellman key share of node i for setting up stream ciphers at the t-th insert operation; $DH_{STONe_i} = h_{STONe}(K_{TC_i} \mid S_{STONe})$
$C^s_{STONe_i}$	Opaque 96-bit capability used for STONe Socket communication to address service s on node i
$ID^{(j)}_{STONe_i}$	64-bit identifier for the j-th virtual hypercube address ($0 \leq j \leq k-1$) of node i; $ID^{(j)}_{STONe_i} = h^{j+1}_{STONe}(DH_{STONe_i})$
$PK^s_{TNS_d}$	Public key in TNS for destination d and service s
$SK^s_{TNS_d}$	Private key in TNS for destination d and service s
$ID^s_{TNS_d}$	Name identifier in TNS for destination d and service s

Table 3.1: Definitions of terms used in the thesis.

Distribution	Symbol	PDF/PMF	Entropy
Binomial Dist.[1]	$B(n,k,p)$	$P(X=k)=\binom{n}{k}p^k(1-p)^{(n-k)}$	$\ln(\sqrt{2\pi enp(1-p)})$
Uniform Dist.	$U(a,b)$	$P(X=k)=\begin{cases}\frac{1}{b-a+1}, & a\leq k\leq b\\ 0, & otherwise\end{cases}$	$\ln(b-a+1)$
Exponential Dist.	$Exp(\lambda)$	$f(x)=\lambda e^{-\lambda x}$	$1-\ln(\lambda)$
Normal Dist.	$N(\pi,\sigma)$	$f(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	$\ln(\sigma\sqrt{2\pi e})$

Table 3.2: Definitions of probability distributions used in this thesis.

[1]The entropy for the binomial distribution assumes the central limit theorem. [173]

3.6.2 Probability Theory

Table 3.6.2 shows the probability distributions we use in this thesis. The entropy $H(X)$ corresponds to Shannon's original formula [168]: For a discrete distribution with $p_i = P(X = i)$ we have $H(X) = \sum_{i=0}^{N} p_i \log p_i$. In the continuous case it is $H(X) = \int_0^\infty f(x)\log f(x)dx$ where $f(x)$ is the probability density function. We say the random variable X follows distribution D when $X \sim D$. To approximate the Binomial Distribution with the Normal Distribution we use Central Limit Theorem [173]: When n is large $B(n,p) \sim N(np, np(1-p))$.

3.6.3 Cryptography

In this thesis we use public key encryption, signature schemes, and hash functions. As in our communication model a message m is a bit string of arbitrary length, and this message for the cryptography operating can be any type of data, within a session or on the network-layer. In $m = m_1|m_2|...|m_k$ we compose a message m from k messages $m_1...m_k$. We denote a message m signed with the corresponding private signing key of node i's public key k_i by $<m>_{k_i}$. The hash of a message m is $h(m)$. The k-times iteration of h on a message m is written as $h^k(m)$. $\sigma, k_i)$ denotes the verification of signature σ using public key k_i. Furthermore, we denote $E_k(m)$ the encryption of message m under key k, and $D_k(c)$ is the decryption of ciphertext c under key k.

Chapter 4

STONe Design

In the previous chapters we introduced Trusted Overlay Networks as a distributed computing architecture that establishes trust between participating nodes. STONe's design is based on a Trusted Overlay Network that consists of distributed trusted proxies. STONe's main design goals are: *scalability*, *resilience* and *resistance against traffic analysis*.

To ensure scalability and resilience STONe has to support short node insertion times, which requires fast neighbor discovery and key exchanges. In contrast, an anonymity network that relies on a static set of proxies, such as Tor [75], does not need to be scalable.

Therefore, we designed STONe as a structured overlay network that is based on a topology similar to a hypercube (see e.g.[147]). Such a topology is more advantageous than a tree or ring structure since it minimizes the number of key exchanges and the average path length at the same time, thus optimizing the network for high churn.

These are the design goals in STONe:

Decentralized Control
: STONe is distributed across different administrative domains. In Trusted Overlay Networks nodes can be administered by different people, but security and anonymity during communication are still guaranteed across the network.

No Central Membership List
: STONe does not have a central membership list of nodes in the system. Each node recognizes only the addresses of its immediate neighbors and any information that should be protected is hidden within the trusted process. A central membership list is a potential threat to anonymity, because an adversary can identify all nodes of the network and shut them down.

Secure Communication
: STONe provides application-level endpoints for a secure communication infras-

tructure. When two parties communicate, an adversary should not have the chance to launch any common attacks unless she breaks into the Trusted Computing hardware.

Transparent Anonymous Communication
: STONe Sockets are application-level endpoints for anonymous communication. When an application uses STONe Sockets, it is hard for an adversary to compromise anonymity of the nodes participating in communication.

Large Address Space to Protect against DDoS attacks
: STONe's overlay network provides a 96-bit address space that contains nodes that are behind different NATs and firewalls. In STONe an attacker is not able to scan the 96-bit address space in a brute-force manner to locate nodes she wants to attack.

Self-Certifying Peer Addresses
: STONe provides a Trusted Name Services (TNS) that maps names to anonymous addresses and self-certifies these addresses in the overlay without revealing identities to anyone. The application has full control over any information that goes public in the overlay.

Secure and Anonymous Routing
: STONe is functional despite common attacks on the overlay. For example, a denial-of-service attack on the overlay should not weaken the system significantly, since an adversary could use this attack to sabotage the anonymity service.

Scalable and Resilient Routing
: STONe provides a scalable and resilient routing infrastructure layer to maintain security and anonymity in the network across firewalls and NATs. In STONe, every node must be able reach every other node with high probability. Furthermore, STONe routers are not always available. Nodes that forward messages in STONe frequently leave and enter the network. Therefore, STONe has to provide a stealthy scheme to keep routing tables up-to-date. This an important feature in a system that provides an infrastructure with different services. For example, in some applications. such as Instant Messaging or Internet telephone, clients have an incentive to stay online for longer time intervals, whereas a simple file-download client may log off after a download finishes. Trusted Overlay Networks' ability to protect against Byzantine faults without using expensive replication techniques provides a strong basis for tolerating high churn in the network.

Simple Programming Interface and Robust Name Service

STONe provides a simple application socket interface. This interface is similar to normal Internet sockets. Further, STONe requires a robust and trusted name service that maps names to opaque network addresses. It is important that this name service is robust against common attacks.

4.1 System Architecture

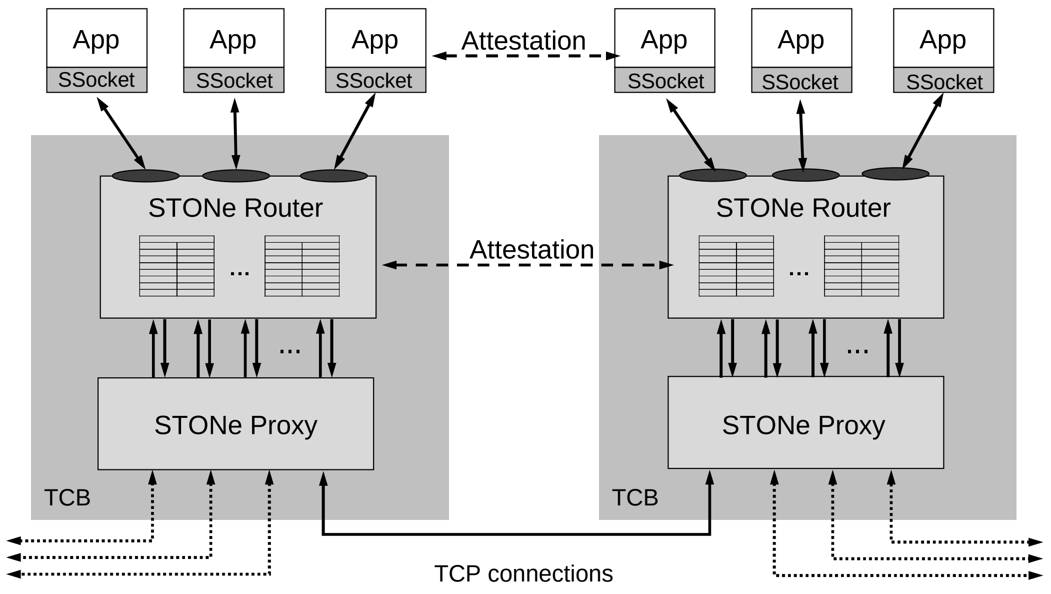

Figure 4.1: STONe Architecture: A single STONe node consists of the Proxy and Router that are located in the Trusted Computing Base of the PC. The STONe Socket library is directly linked to the application. Applications can be inside and outside the TCB.

Figure 4.1 shows the architecture of STONe. STONe does not require any trusted third parties for establishing secure routing, and it also does not require costly traffic analysis protection to ensure anonymity and improve security.

Each STONe node consists of three components: STONe Proxy, STONe Router, and the STONe Socket library, as shown in figure 4.1. The STONe Proxy and the Router are two individual processes located in the TCB. When a node connects to STONe it proves by remote attestation to its neighbors in the overlay that the contents of the TCB – STONe Proxy and Router – are trusted, and that neighbors in the overlay can trust future transmissions from this node. Applications are completely independent from STONe. If they run inside the TCB, a STONe node can use remote attestation as well to verify trustworthiness of the application to its peers. Instead of linking a standard OS socket library, applications have to use the STONe socket library to communicate with other applications in the overlay.

Network Topology STONe is a structured routing overlay network derived from a hypercube topology that not only provides scalability but also availability across firewalls or NATs. This is a fundamental difference to content-distribution overlays such as Gnutella [7], where the the total amount of data in transit in the overlay network is much less, because it only serves lookup requests and no data transfers. STONe is also different from web caching overlay networks [1], because nodes may enter and leave the network, and participating nodes are not necessarily located in the public Internet but in private networks behind NATs.

In general, STONe works on any structured overlay. However, STONe has to optimize the overlay structure for scalability and resilience. STONe's topology is equivalent to a CAN network with diameter $d = \log N$, but as pointed out earlier, the main difference to many existing structured peer-to-peer networks is that every STONe node itself is the key and vice versa. Hence, STONe does not require leaf nodes that replicate objects, as for example in Pastry [154]. But it requires redundant routing paths, since nodes may frequently leave and enter the overlay. Furthermore, latency has to be as short as possible, and the overlays's goal is to minimize the number of hops on the routes and the routing table size should be minimal.

A node joins STONe by authenticating itself to an existing *bootstrap* node on the overlay using remote attestation, and from there it finds its existing neighbors on the hypercube. The node's hypercube address in the overlay network is a cryptographically-secure keyed hash of its Diffie-Hellman key share: $ID^{(j)}_{STONe_i} = h_{STONe}(DH_{STONe_i})$. DH_{STONe_i} is derived from the Trusted Computing secret $K^j_{STONe_i}$. Assigning addresses in this fashion randomizes the overlay topology and makes the system more robust against traffic analysis. The built-in secret Trusted Computing key K_{TC_i} is secured in hardware making it hard for an adversary to forge a valid identity.

STONe Proxy: Similar to proxies in other anonymity networks, the STONe Proxy is responsible for relaying packets between adjacent STONe nodes. The Proxy maintains for each of its neighbors a connection state and a shared stream cipher with the shared keys $K^j_{STONe_i}$ between nodes i and j.

Whenever the STONe Proxy connects to another node in the network, it first does a TCP handshake, then executes the remote attestation protocol, and finally runs a simple Diffie-Hellman key exchange to set up a shared key. STONe keys are automatically certified by remote attestation.

The STONe Proxy protects STONe against attacks from the underlying network. An attacker can launch attacks on STONe from outside the overlay by injecting packets into the TCP streams or dropping packets in some streams. The STONe Proxy acts as a rudimentary firewall; packets arriving at wrong ports, or with invalid TCP sequence numbers are silently dropped. When the STONe Proxy identifies too many corrupt packets on one stream, it quickly establishes a different TCP connection with its peers.

If an attacker overloads a STONe Proxy by flooding it with packets, the router would normally drop TCP connections. Yet, when these packets only initiate new connections it would still prevent the router from processing other packets without additional delay. In STONe we separate Proxy and Router, and therefore the Router is able to start a different STONe Proxy and reconstruct the connection state using the information from the routing tables in the STONe Router. This is not the only advantage of the separation between the STONe Proxy and Router. It also allows encryption and routing of packets to happen in parallel. In particular, future commodity PCs will have symmetric multiprocessing capabilities, and therefore encryption and routing can take place in parallel.

4.1.1 STONe Router

The STONe Router is the main part of the system. It maintains the routing tables, routes STONe packets in the overlay, and provides STONe sockets to the application. The Router also handles the initial handshake for new nodes entering the system. A node that wants to join STONe connects to the Router, after it has successfully completed remote attestation. To protect against attacks from the network, the Router does not accept any messages from nodes that did not succeed in remote attestation. After the new node has connected, the Router hands off the connection to the STONe Proxy. When application data arrives from the STONe Socket Library, the Router creates a STONe packet and forwards it to the STONe Proxy. When STONe packets arrive from the network via the STONe Proxy, the Router looks up the next hop in its routing table, and forwards the packet to the STONe Proxy.

The STONe Router provides two service abstractions to the applications through the STONe Socket library— a connection-oriented trusted stream service (TSS) and a trusted datagram service (TDS). The internal protocol in TSS is identical to TCP, but it implements only flow control, because congestion control would interfere with the underlying connection. Instead, TSS relies on the congestion avoidance mechanism that is inherent in our anonymous routing technique, as we will see in the next sections. In contrast, TDS provides a connection-less datagram service similar to UDP. There is no extra protocol header overhead. TDS and TSS use the fields in the STONe fragment that we describe now.

4.1.2 STONe Packets and Fragments

STONe packets – the communication unit for datagram sockets on the trusted datagram service (TDS) – consist of *multiple STONe fragments*. Fragments and packets have the same format, with packets being the application-level unit. Figure 4.2 shows the

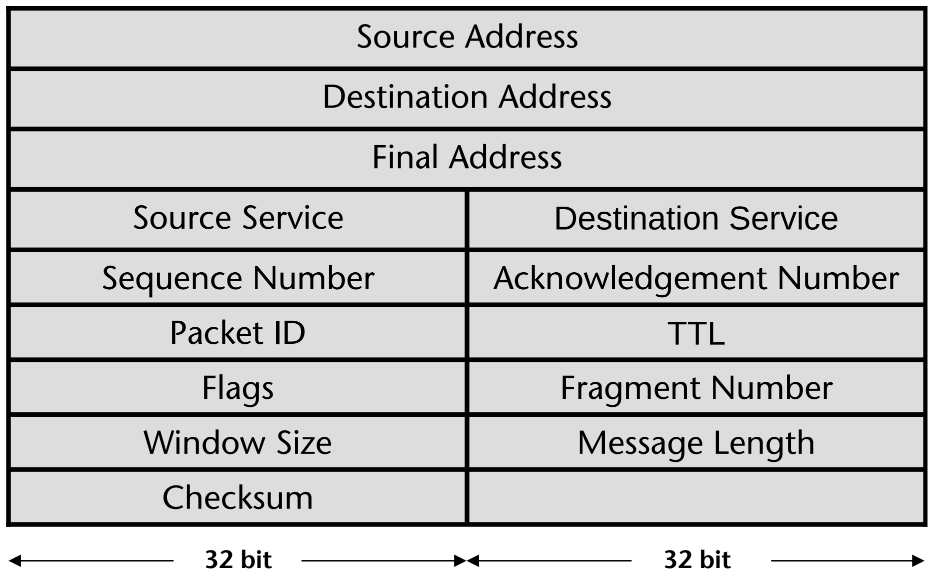

Figure 4.2: STONe fragment header: This 68-byte header is mostly self-explanatory. STONe addresses are generally 64 bits long. Flags describes special fragment types such as echo fragments, that are used for end-to-end delay measurements.The Packet IDs are listed in Table 4.1.

STONe fragment header. It contains three 64-bit STONe addresses, the *source*, *destination* and *final* addresses, and the corresponding 32-bit service-port numbers. First we explain routing that does not make use of the final field, which does not protect against Traffic Analysis. The STONe node sending the fragment places the destination address in the *destination* field. The final field is reserved for the use of an intermediate relay when STONe protects against Traffic Analysis. Other fields are self-explanatory and similar to TCP/IP. For example, the STONe Packet ID marks the content of the packet as either a control packet or a user-data packet. And the source and destination *services* are used for multiplexing different socket endpoints similar to TCP/UDP ports. The size of a STONe fragment can be variable and does not depend on IP packet size constraints because STONe is based on TCP, which is a stream protocol without fixed packet sizes.

When the Router receives a fragment from the Proxy, it computes the checksum over the whole fragment. Then the Router determines the TTL of the fragment. When the TTL of the fragment is 0 the fragment is dropped – otherwise the Router decrements TTL and uses STONe's routing algorithm to look up the next hop. Finally, the fragment with the updated checksum gets forwarded to this new hop.

Message Types Table 4.1 shows the different message types for the Packet ID field in the STONe header. Most of them are self-explanatory and relate to TCP's packets. A joining node sends an T-INSERT-KEY message into the network. When the T-INSERT-KEY arrives at the destination node it returns a T-START message. Data packets have a

Type	Description
T-SYN	initiates handshake with another node
T-SYNACK	acknowledges T-SYN packet
T-ACK	acknowledges message packets other than T-SYN
T-INSERT-KEY	sends an insert key message into the network
T-START	confirms when insert key message arrives at correct node
T-DATA	sends STONe data messages
TSS-CONNECT	initiate a TSS connection
TSS-FIN	terminate a TSS connection

Table 4.1: Packet Types in STONe: Types starting with a T are for low-level communication in STONe, and TSS packet types are for high-level communication in the TSS abstraction.

T-DATA identifier. STONe does not require a T-FIN packet, because FINs are implicit through TCP. Whenever the TCP connection with another node terminates, STONe also terminates the connection automatically.

4.1.3 STONe Socket Library

The STONe Socket Library is a wrapper that exports a standard socket interface for backward compatibility with existing applications and ease-of-use. Internally, each socket file-descriptor is mapped onto a 96-bit opaque capability. The socket system calls are trapped by the library and mapped onto messages over UNIX sockets to the local STONe router. The library also implements a pseudonym look-up service, i.e., a client for the Trusted Name Service (TNS) that maps names to self-certifying capabilities. STONe is hardened against attacks on the socket library. Even if an attacker were to modify the library, she would not be able to access confidential STONe information or compromise the anonymity of communication.

4.2 Secure Communication

Ideally, secure communication would be provided by the Internet protocol, but unfortunately, there are some common problems. Even though security protocols like IPsec, TLS, OpenSSL or OpenSSH provide network-layer and end-to-end security [108, 100, 73, 15, 14], secure routing is still an unsolved problem. Routing protocols like OSPF distribute link-state on the Internet [130], but in environments that are prone to Byzantine failures these routing update messages are still subject to attacks [142]. On the other hand routers can be malicious [128].

Furthermore, network address distribution must be secure. An adversary who is able to obtain large number of network addresses is able to compromise the network. This is especially a problem in modern network configurations where addresses are obtained dynamically. For example, in a peer-to-peer network an adversary can often obtain as many nodeIDs as she wants and launch a so-called 'Sybil attack' [56].

STONe also has to secure the overlay network and make it resilient for random routing. Random routing is designed to work in a homogeneous, fast, and reliable environment such as workstation clusters, and in STONe random routing is used to work over a heterogeneous and unreliable Internet.

4.2.1 Securely Assigning Network Addresses

Assignment of network addresses is a common security problem, because an adversary can imitate other nodes to compromise the network. On the Internet, IP addresses are usually allocated by the network provider – often dynamically – and they cost money. In contrast, in peer-to-peer networks, an adversary running a large number of router instances can also obtain a separate address or NodeID for every node in the network and compromise the network. In the Sybil or Eclipse attack malicious nodes compromise the peer-to-peer network and imitate other nodes to intercept communication [78]. Certificates are always an option to ensure the validity of NodeIDs, but they require a central trusted certificate authority, which is not compliant with STONe's scalability requirements. In STONe it is important for routing to use multiple NodeID and have redundancy, but on the other hand we want to bind the number of NodeIDs to the number of available nodes in the system. For example, an adversary could degrade performance of the system by adding a slow machine and assign this machine a vast amount of addresses. Using IP addresses as NodeIDs is also not desirable because they are not unique across NATs and firewalls.

Trusted Overlay Networks already have built-in secret keys K_{TC_i} that are unique and certified on every platform. Initially, STONe sets up the Diffie-Hellman key share $DH^t_{STONe_i}$ from the built-in AES key K_{TC_i} using a secure hash function h_{TC}. The key share $DH^t_{STONe_i}$ is used on this node to set up an encrypted tunnel between i and its neighbor j that uses key $K^j_{STONe_i}$.

STONe computes the first NodeID $ID^{(0)}_{STONe_i} = h_{TC}(DH_{STONe_i})$ from the hash of this key share. Computing the NodeID from DH_{STONe_i} has multiple advantages:

(i) Any of i's *neighbors can verify* that i really has the key share, otherwise it would not be able to decrypt the data on the secure channel. Therefore, a neighbor can verify that node i indeed uses the correct NodeID.

(ii) *STONe arranges nodes randomly and independent of their location* on the underlying Internet topology. This makes it hard for an adversary to insert nodes close to a specific node in order to eavesdrop on communications or DDoS the node.

(iii) STONe *limits the number of NodeIDs per Trusted Computing platform* (i.e. for every secret key) because STONe generates the key share from the internal secret AES key K_{TC_i} and some randomness. The only way for obtaining more NodeIDs is to buy more hardware or break existing hardware outside of the current STONe network. A second router on the same platform would only be able to acquire the same set of NodeIDs. Nevertheless, one router per platform does not restrict the number of socket applications because the STONe address uses 32 bits for multiplexing different applications.

4.2.2 Secure Routing

The optimal routing geometry for STONe would be a fully connected network in which every node can reach every other node within a single overlay hop that is the shortest distance between the two nodes on the Internet. This approach, however, is not scalable because a node joining the network needs to obtain identities and keys from all other nodes in the network. This causes a large time overhead, especially because key changes are expensive. Furthermore, it uses too many ports on the machine – this is a common problem when it is located behind a firewall. Therefore, STONe uses a structured routing overlay for scalability and reliability. However, STONe as a routing overlay differs from DHT-style routing in many ways:

(i) STONe's *unique node addresses are the only keys*. No additional or replicated keys exist.

(ii) STONe has to *minimize latency* in the network and not only maximize throughput as in content distribution.

(iii) STONe *requires alternate routes* for fault-tolerance instead of replicas.

Because STONe is based on Trusted Overlay Networks it has a Fail-Stop failure model, compared to the Byzantine model of other comparable overlay networks. A STONe node that fails or crashes does not turn malicious unless the Trusted Computing hardware gets compromised. Adjacent nodes will detect the crash because the kernel network stack sends a FIN packet in most cases, unless the machine suddenly gets disconnected from the network for some reason. To cope with this situation and check whether a connection is still alive STONe has to send probes over the connection regularly if it doesn't expect any real data [30, 159].

Therefore, STONe can be *self-organizing* and *self-maintaining* and does not need to protect against traitors as compared to systems that have Byzantine failures. Since nodes are trusted in Trusted Overlay Networks and do not suffer from such Byzantine failures, STONe uses *oblivious routing* (i.e. the next hop is determined by the destination and the current hop only) and also *dynamic routing* (i.e. it selects the next hop from the set of possible next hops based on lowest cost). STONe uses a *hypercube-similar topology* because hypercube routing is efficient and requires only $O(\log N)$ routing table entries for an overlay path length of $O(\log N)$. STONe needs to take into consideration routing table size in addition to path length because nodes have to connect to their neighbors when they join the network. Also, hypercubes are symmetric and balanced. This is important for providing anonymous communication that protects against Traffic Analysis. Traffic Analysis in an asymmetric topology can be significantly easier because traffic on bottleneck links between two network partitions gives clues about the communication patterns.

However, STONe does not use the Internet address space but creates its own for several reasons: First, nodes behind different NATs may have the same private IP addresses. Furthermore, in a 32-bit IP address space it is easy to launch DDoS or probing attacks on nodes by random guessing, even if routing is anonymous [42]. Therefore, STONe has to use an extended uniform address space. Internally, STONe addresses are 64 bit long with a 32 bit service ID. But externally, they appear to STONe applications as 96 bit opaque capabilities.

STONe uses static routing tables that only depend on the geometry of the virtual network topology. But they also allow dynamic routing decisions based on cost metrics within the static structure. Internet routing protocols such as OSPF [130] update dynamic routing tables periodically to optimize routes and propagate link failures. But secure dynamic routing is a hard problem because (i) routing updates have to be *trusted for correctness* and not only for performance, (ii) an adversary can *simulate link congestion* to be able to receive more traffic for routing than the other nodes, and (iii) it is *hard to maintain a balanced routing topology* when routing tables get updated dynamically, which is bad for anonymity. For example, BGP is prone to attacks, since it does not fulfill any of the three issues mentioned [97].

Static routing tables, in contrast, have stronger security because routing tables only depend on NodeIDs and the routing geometry, but performance may not be optimal. STONe balances the static topology by determining NodeIDs independently from their geographic location. If an adversary wants to inject a node close to another specific node, she has to obtain lots of NodeIDs and therefore purchase lots of Trusted Computing hardware. Static routing tables also give guarantees for path lengths or routing table sizes. However, static routing tables doe not optimize for performance and scalability inherently and needs to be reorganized in case of link failures. STONe's static routing tables are derived from an approximation of a hypercube that we discuss next.

Hypercube Properties A typical hypercube is a graph $G = (V, E)$ with vertices V and edges E. It is the generalization of a three-dimensional cube to d dimensions. Every vertex of such a hypercube has d edges, and in total a hypercube has $N := 2^d$ vertices. Addresses on hypercube vertices are bitstrings, and adjacent vertices always have Hamming distance 1. A hypercube is a recursive structure. By connecting two $(d-1)$-dimensional hypercubes we can build a d-dimensional hypercube. In this new d-dimensional hypercube the vertices' addresses of the two $(d-1)$-dimensional hypercubes are extended by 1-bit prefix 0 or 1, depending on the hypercube the vertex is from. The Hamming distance between two arbitrary nodes defines the distance in the network.

We call a *matching* of a graph a set of edges without common vertices. A matching is perfect when the matching covers all vertices. A hypercube has a perfect matching, and this means that we can split the hypercube in two halves.

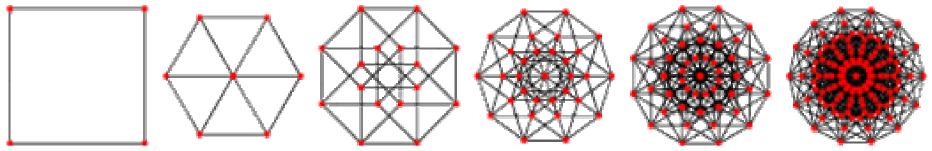

Figure 4.3: Projections of 2- to 7-dimensional Hypercubes from left to right (Source: MathWorld)

Figure 4.3 shows projections of multi-dimensional hypercubes onto the two-dimensional space. One big advantage of a hypercube is its large bisection width of $\frac{N}{2}$ in a network of N nodes compared to other topologies. This eliminates many congestion bottlenecks and makes it highly scalable. Furthermore, hypercubes have short path lengths. The diameter is only $\log N$ [120].

Standard hypercube addresses consist of bit strings, and on every edge a different bit is flipped such that two adjacent node addresses have Hamming distance 1. Routing is then similar to class-less routing in CIDR [85]: The node compares the bit string of the destination address with the one of the current address from left to right and forwards the message to the node with the first difference.

When we pick two random nodes from anywhere in the hypercube and compute the Hamming distance between the random nodes, this distance has a Binomial distribution that depends on the number of nodes in the network: A hypercube's address length is $d = \log N$ bits, and every bit is 0 or 1 with probability $\frac{1}{2}$. This is a Bernoulli experiment with $\log N$ trials and probability $p = \frac{1}{2}$. Accordingly, we derive the probability $P_{dist}(N,k)$ that the path length between sender and the receiver over a hypercube of N nodes is k as follows:

$$P_{dist}(N,k) = \frac{1}{N}\binom{\log N}{k}.$$

The expected distance is $\mu_{dist} = \frac{\log N}{2}$, and the maximum distance is $M_{dist}(n) = \log N$.

STONe uses hypercube-based routing, since hypercube routing is highly scalable,

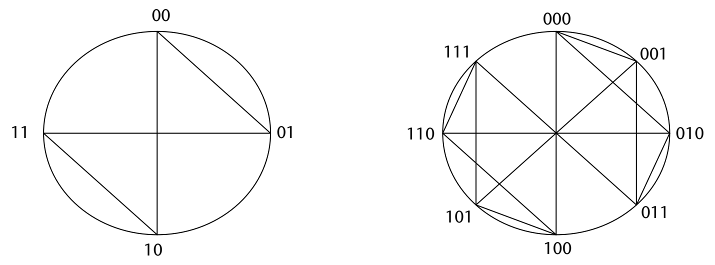

Figure 4.4: Ring versus Hypercube Routing for 2 dimensions (left) and 3 dimensions (right).

symmetric and has short path lengths. The main difference, however, is that not necessarily all nodes of the hypercube may exist in STONe. The hamming distance between two neighbors in STONe could therefore be greater than 1.

In a hypercube network of N nodes a single node has about $\log N$ routing table entries similar to a CAN with dimension $d = \log N$ [147].

Figure 4.4 shows the difference between a ring and a hypercube. In content-distribution networks ring or tree topologies have more suitable properties than hypercubes when frequent failures occur in the network [92], but they have a low bisection width and therefore worse behavior under congestion. Routing in content-delivery networks solves congestion by replication, but routing architectures like STONe have to be optimized for alternate routes in case of failures. And routing messages in a ring can become inefficient for message forwarding on an alternative route even when the path length is still $O(\log N)$ [180]. For example, when a message for $1\underbrace{1...1}_{N-1}$ happens to be at $10\underbrace{1...1}_{N-2}$, and this last link is broken, it goes to $11\underbrace{0...0}_{N-2}$, and then it needs to fix N-2 bits again with an overhead of $O(\log N)$.

Hypercube-based Routing in STONe Because STONe's NodeIDs are randomly distributed across a fixed 64 bit address space, STONe cannot use standard hypercube routing with bit fixing from left to right. Even with consecutive NodeIDs, hypercube routing is not possible, since nodes enter and leave the network constantly, and there are always some address gaps in the hypercube.

Instead, STONe uses prefix-based routing on the partial hypercube. In every slot i of the routing table STONe stores the node that matches i bits of the address' prefix. When there are multiple addresses that suit this requirement STONe picks the one with the closest Hamming distance, as this optimizes the route.

During routing on every hop the prefix length increases, and eventually, the fragment reaches its destination. When the prefix length increases, the Hamming distance may decrease or increase depending on the remaining bits after the prefix, but as the prefix length increases the Hamming distance also decreases.

When the best prefix match returns a node that does not make any progress, either the hypercube is broken or the destination does not exist. What STONe should do in this case depends on the routing semantics: If the destination address exists for sure it tries to forward the packet to the next of the k destination addresses, thus picking an alternative path. If it is an *insert node* operation, and the destination address does not exist in STONe, it handles the fragment at this node.

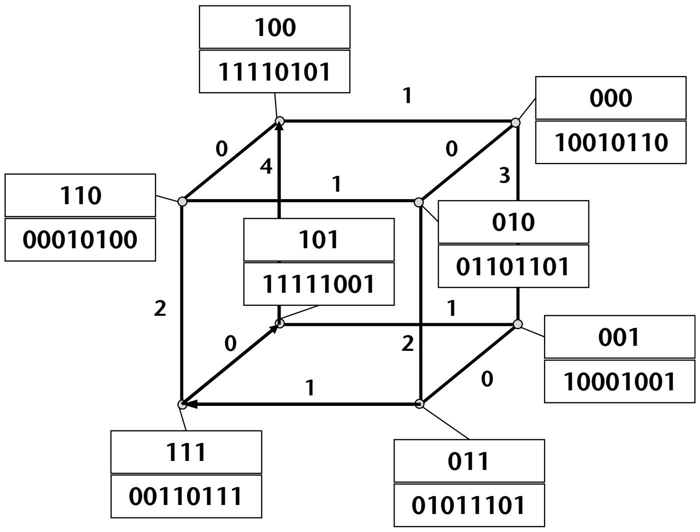

Figure 4.5: Hypercube Routing in STONe with 8-bit identifiers (b=1 and k=1): The top address is a standard 3-bit hypercube address, whereas the bottom address of the label is an 8-bit STONe address. The number on the edges is the common prefix length of the two adjacent nodes.

Figure 4.5 shows hypercube routing in STONe compared to standard hypercube routing. In this example every node has a random 8-bit long NodeID. In normal oblivious hypercube routing the current node XORs the destination address with its own address and forwards the message to the node that fixes the leftmost bit that is not zero. In STONe not every bit in the address has to be fixed, so we route by prefix length, i.e. the bit position in the routing table where the leftmost bit is different from the destination address.

Prefix-based hypercube routing is optimal. Given N n-bit addresses the expected path length is always less than $\frac{\log N}{2}$, which is the same as an N-dimensional hypercube. Using prefix-based routing tables we note that the Hamming distance to the destination on the first i bits is 0 after i hops, but the Hamming distance of the whole bit string may increase temporarily along the path. However, the neighbors of a node are not always the closest in their Hamming distance. For example if STONe has nodes 00000, 10100, 11010, 11011, and 11111, and it routes from 00000 to 11111, it traverses through 10100, 11011 to 11111. However, 11010 has a smaller Hamming distance to 11111 than to 10100 and is not in 11111's routing table, which is already used by 11010 when $l = 1$.

Optimizing Resilience The advantage of a static routing topology like STONe is that it is resistant to malicious or bogus routing updates. However, static hypercubes have the disadvantage that a single link failure already breaks one complete path. When STONe detects a link failure it usually updates its routing tables and sends a routing update to its neighbors. However, when routing updates take place too often it degrades STONe's performance and scalability significantly. And often congestion or failures are only temporarily and link quality may improve again after a short time. This also protects against temporary DDoS attacks that can be fixed after a short time.

Additionally, a single link failure breaks $O(\log N)$ paths in the hypercube, and when a single link fails along the path, the whole path is broken. Therefore, given our routing strategy in a simple hypercube with a fraction of p_{IP} faulty Internet paths, the probability that a path of length L_{path} fails is

$$P_{fail}(N, p_{IP}) = 1 - \frac{\binom{(1-p_{IP})N}{L_{path}}}{\binom{N}{L_{path}}}.$$

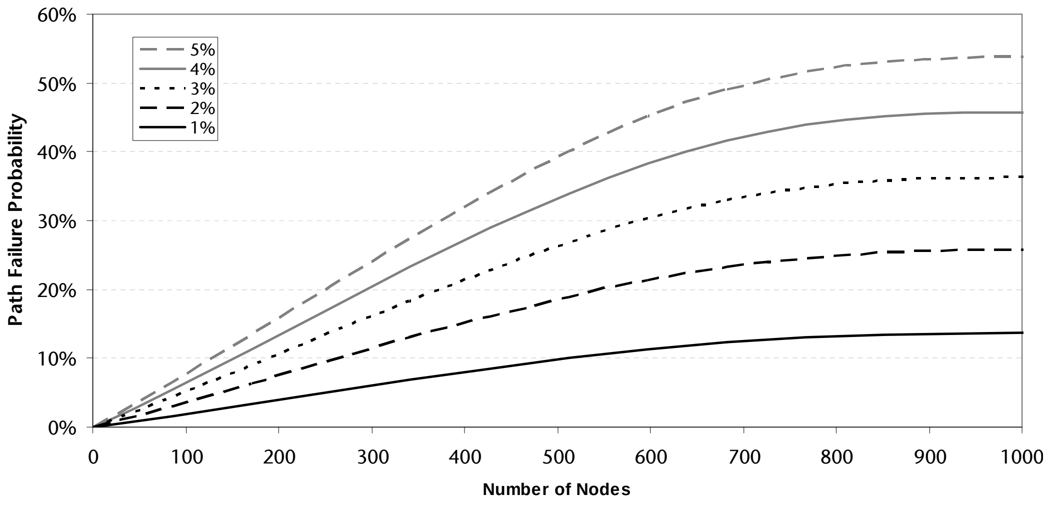

Figure 4.6: Probability P_{fail} that a single message transmission along a hypercube path fails under different Internet link failure rates $1\% - 5\%$ when $l = 1$ and $L_{path} = 2 \log N$.

Figure 4.6 shows P_{fail}, when the individual link failure p_{IP} varies from 1% to 5%. STONe, therefore, maintains l alternative links per entry in a hypercube node to handle short-term failures. When the link quality is bad STONe picks a different link. STONe maintains a reliable connection between the nodes, and each node knows almost immediately when its peer goes down, but a STONe node can experience internal congestion and drop packets. When $l > 1$ P_{fail} is almost P'_{fail} except that p_{IP} gets replaced by

$$p'_{IP} = (1 - p_{IP}) \sum_{j=0}^{l-1} p_{IP}^{j}.$$

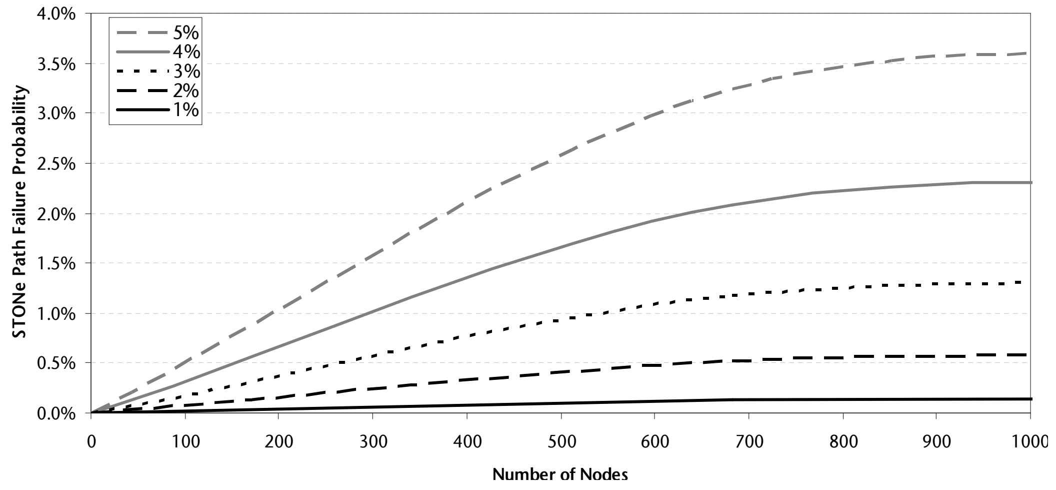

Figure 4.7: Probability P_{fail} that a single message transmission along a hypercube path fails under different Internet link failure rates $1\% - 5\%$ when $l = 2$ and $L_{path} = 2 \log N$.

Figure 4.7 shows the path failure probability P'_{fail} for different Internet path failure rates. We can assume that path failure rates between 0.9% and 1.9% are realistic [139], and under STONe's redundant hypercube with $l = 2$ the network is able to maintain the same Internet path failure rate for STONe paths.

Optimizing Bandwidth The hypercube may not only have bottlenecks or congestion, but because of the static hypercube on a heterogeneous network some nodes may be underutilized and some overutilized. To make the overlay more homogeneous, STONe uses up to k virtual addresses per node to separate low-bandwidth from high-bandwidth nodes. Nodes with higher bandwidth receive more traffic on the average than nodes with low bandwidth. Furthermore, nodes have alternative paths to pick from when there is a failure or bad link quality. When all nodes use $k = \frac{N}{\log N}$ STONe behaves like an unstructured network. STONe computes the $k-1$ additional addresses from the private key in the Trusted Overlay Networks of the node by *chaining the hash function*:

$$ID^{(k)}_{STONe_i} = h^{k-1}_{STONe}(DH_{STONe_i})$$

For optimal bandwidth distribution, every node gets k_i virtual addresses. If every node i has k_i virtual addresses then the probability that node i is on the path is

$$P_{path}(i) = 1 - \frac{\binom{K-k_i}{\log N}}{\binom{K}{\log N}} = 1 - \frac{\binom{N-1}{\log N}}{\binom{N+R}{\log N}}$$

where $K = \sum_{i=1}^{N} k_i$.

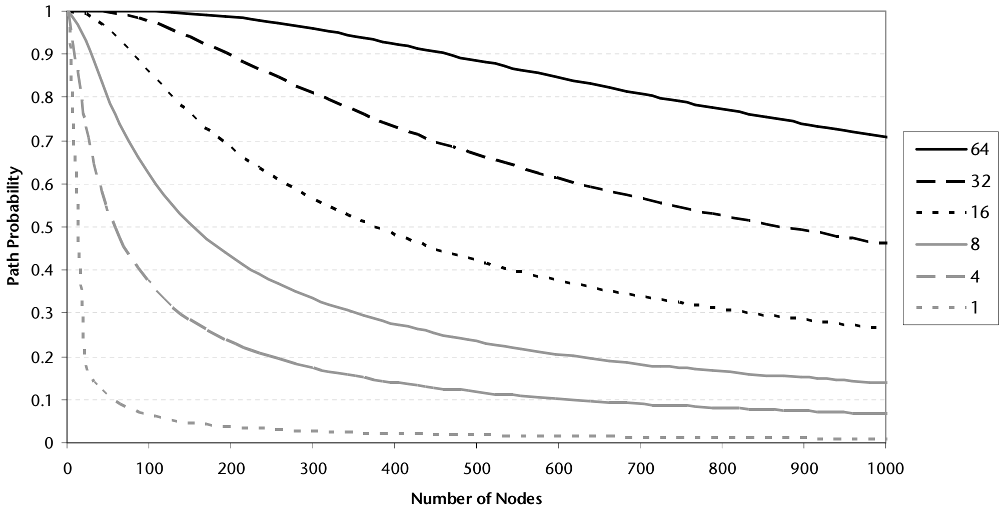

Figure 4.8: Probability $P_{path}(i)$ that a specific node i is part of a random path in random routing on STONe for different $k_i = R\bar{k}$: The larger R gets compared to average $\bar{k}$ the larger the probability is, but it decreases exponentially depending on the number of nodes in the network. In this figure $\bar{k} = 1$ and $R = 1, 4, 8, 16, 32, 64$ depending on the graph.

Figure 4.8 shows the probability distribution for different ratios $R = \frac{k_i}{\bar{k}}$ with $\bar{k} = \frac{K}{N}$ and $K = 2N$, and in this case $\bar{k} = 1$. When we increase the parameter R, a high bandwidth node has a better chance to be on an arbitrary path. The probability decreases exponentially with the number of nodes in the overlay network.

Optimizing Path Length STONe optimizes the hypercube further: To reduce the number of hops along a path STONe is not based on a Boolean hypercube. Instead it picks a different base b for the logarithm which shortens the path length exponentially. This is similar to Pastry [154] where b determines the number of columns in the routing table. Pastry uses a tree geometry that is not suitable for STONe, since STONe is a routing overlay and does not have any leaf nodes. The downside is that Pastry also increases the routing table size linearly. Only for $b = N$ STONe's overlay network is a fully connected network.

In the optimized hypercube the Hamming distance is then defined as the Manhattan distance, ie the sum of the absolute differences of the single digits. Otherwise the hypercube insert algorithm is the same, except that the routing table updates have to be sent to all nodes with the same minmal Manhattan/Hamming distance.

Considering the optimizations for resilience, bandwidth, and path length in the hypercube, STONe's *routing table size* is

$$S(N) = \bar{k}l(b-1)\left(\log_b(\bar{k}N) - \frac{l-1}{2}\right).$$

The correction factor $(1-\frac{l-1}{2})$ is required, because there are not enough prefix matches for the last slots. The $(\log_2 N)$-th slot has only one match, whereas the $(\log_2 N - 1)$-th slot has two matches and so on. Therefore, we need to subtract $\sum_{i=0}^{l} i = \frac{l(l-1)}{2}$ from l.

The *expected path length* for two random addresses in a normal Boolean hypercube is $\frac{\log_2 N}{2}$, but in STONe it is only

$$L(N) = \frac{1}{2}\log_b\left(\frac{N}{\bar{k}l\left(1-\frac{l-1}{2\log_b(\bar{k}N)}\right)}\right).$$

.

Cost-based Routing In addition to the prefix length and the Hamming distance, STONe uses a cost metric to allow dynamic routing decisions to take place when link failures or congestion occurs. This cost metric depends on the link quality. When a STONe packet x arrives at a node, the router computes the longest prefix match with x, and looks up the slot in the routing table that has this prefix match. It then searches for the entry with the best cost metric out of the l entries in each slot and forwards the packet to this hop.

The cost metric is $\frac{c(x)}{d(x,y)}$. $c(y)$ is the link quality to next hop y, and $d(x, y)$ is the Hamming distance between the destination address y of the packet and the next hop x. Because the longest common prefix is the main routing criterion, STONe makes sure that packets arrive in $\log N$ steps and do not circulate in the network. Algorithm 2 shows the routing algorithm. `Cost` is the cost function, which is the link quality metric divided by the Hamming distance of the routing table entry to the destination. The function `Prefix_length` returns the common prefix length of the two addresses. First, the algorithm picks the longest prefix match out of the k virtual addresses. Then it looks up the best match with the destination address given the best match out of the k addresses. If the next hop does not increase the prefix match, the algorithm tries the virtual address with the next best prefix match and so on.

4.2.3 Secure Maintenance

Maintenance in STONe has three functions: Handshake, Neighbor Discovery, and Routing Updates. When a new node joins it has to follow STONe's handshake protocol

Algorithm 1 Send Packet

```
send(p, dest)

p.dest= dest
route(p, dest, last)
```

Algorithm 2 Routing Table Lookup

```
route(p, dest, last)

max_slot= 0
for $i$ = $0$ to $k$ do
  a[$i$]= prefix_length(local[$i$], last)
end for
sort(a)
for $j$ = $0$ to $k$ do
  max_slot= a[$j$]
  min_cost= MAX_COST
  for $i$ = $0$ to $k$ do
    slot = prefix_length(local[$i$], dest)
    if slot $>$ max_slot then
      max_slot= slot
      dest_table= $i$
      for $j$ = $0$ to $l$ do
        cost = cost(slot, $j$)
        if cost $<$ min_cost then
          min_cost= cost
          dest_idx= $j$
        end if
      end for
      exit
    end if
  end for
end for
forward(p, local[dest_table][dest_idx])
```

until it finds the right position in the overlay. Nodes have to constantly discover neighbors in the hypercube and also update routing information based on their TCP status.

Handshake Initially, when a node wants to join STONe it picks a local node, finishes remote attestation, and sets up the shared key. After that, communication with *any node* is encrypted to avoid detection, except for transport-protocol handshakes. The assumption, of course, is that the local node has not been compromised, but in this section we will mention some strategies for detecting this.

When a node connects to the STONe network it first establishes a transport-layer connection with a *STONe bootstrap node*, authenticates itself to STONe by performing remote attestation, and finally sets up a shared key with adjacent STONe nodes. Figure 4.9 depicts, in detail, the handshake when Node 1 connects to Node 2.

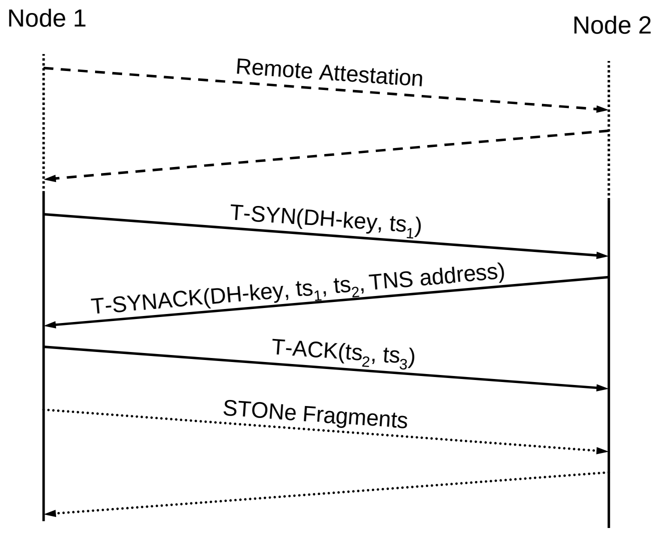

Figure 4.9: STONe handshake protocol: Similar to TCP, STONe's handshake protocol is a three-way handshake protocol. The node that joins the network, Node 1, first sends a T-SYN packet. The bootstrap node, Node 2, then replies with a T-SYNACK, and finally, Node 1, confirms the handshake with an T-ACK packet.

First, two nodes run the protocol for remote attestation in Trusted Computing to make sure they can trust that both are running the same STONe software. If remote attestation fails, Node 2 interrupts the handshake process and filters any further requests from Node 1 for some time interval to prevent DDoS attacks. Otherwise, Node 1 initiates the connection by sending a T-SYN packet to Node 2. The T-SYN packet contains Node 1's Diffie-Hellman key share signed with its signature key, the certificate for Node 1's public verification key, and a timestamp ts_1 when the packet has been sent. Node 2 returns a T-SYNACK packet that contains its Diffie-Hellman key share signed with its signature key, the certificate for Node 2's public verification key, a timestamp ts_2, and Node 1's timestamp ts_1. Node 2 then computes the k STONe addresses $ID^{(j)}_{STONe_1}$ of Node 1 using the built-in hash function h_{TC} and shared keys $DH^t_{STONe_1}$, such that $ID^{(j)}_{STONe_1} = h_{TC}(DH^t_{STONe_1})$ for $j = 1..k$. When the T-SYNACK packet arrives at Node 1 it returns a T-ACK packet with timestamp ts_3 of the T-SYNACK packet. Both nodes can compute the latency of the link using the three timestamps. This is the cost metric in the routing table. The cryptographic primitives in the handshake protocol are the following:

Handshake(i, s, S_{STONe})	
j $\leftrightarrow$ i:	• TCP Handshake with j
i:	• Diffie-Hellman key share for node i: $DH_{STONe_i} = e^{x_i}$, $x_i = h_{STONe}(K_{TC_i} \mid S_{STONe})$
i $\rightarrow$ j:	• $(DH_{STONe_i}, sign_{SK_{TC_i}}(DH_{STONe_i}))$
j $\rightarrow$ i:	• $(DH_{STONe_j}, sign_{SK_{TC_j}}(DH_{STONe_j}), k_i)$
j:	• Verify $sign_{SK_{TC_j}}(DH_{STONe_j})$ using PK_{TC_j} • $ID^{(l)}_{STONe_i} = h^{l+1}_{STONe}(DH_{STONe_i})$, $l = 0...k_i$ • $K^s_{STONe_i} = (DH_{STONe_j})^{x_i}$
i:	• $ID^l_{STONe_j} = h^{l+1}_{STONe}(DH_{STONe_j})$, $l = 0...k_j$ • Insert $ID^l_{STONe_j}$ into routing table, $l = 0...k_j$

The authentication mechanism ensures that both nodes have received information about the authenticity of the STONe router software and that both can compute the shared Diffie-Hellman key for exchanging subsequent STONe packets.

Neighbor Discovery After a joining STONe node has completed a handshake with the bootstrap node it has to find its neighbors in the hypercube. Node 1 needs to find its neighbors in the structured overlay. Figure 4.10 shows the insert process: Node 2 forwards a T-INSERT-KEY packet to the address that is closest to Node 1's using the packet forwarding algorithm described in the next section. When the packet arrives at the destination, Node 3, it connects to Node 1, and after the STONe handshake it connects to Node 1 via T-SYN and sends a T-START packet to Node 1. Node 1 and 3 then update their routing tables, and Node 1 learns about its neighbors from Node 3. When the T-INSERT-KEY or T-START packet gets lost the node times out after a while and restarts the joining process. When Node 1 is behind a NAT but Node 3 is not, it forwards the T-START packet through the network back to Node 2 and lets Node 1 connect to Node 3. If both nodes are behind NATs, Node 1 first tries to use Node 2 as an intermediate proxy to set up the connection. If that fails it picks a random node in STONe until it successfully connects to Node 3. Node 2 repeats the insertion procedure for all k virtual hypercube addresses, and therefore it takes $O(k \log N)$ steps in the hypercube.

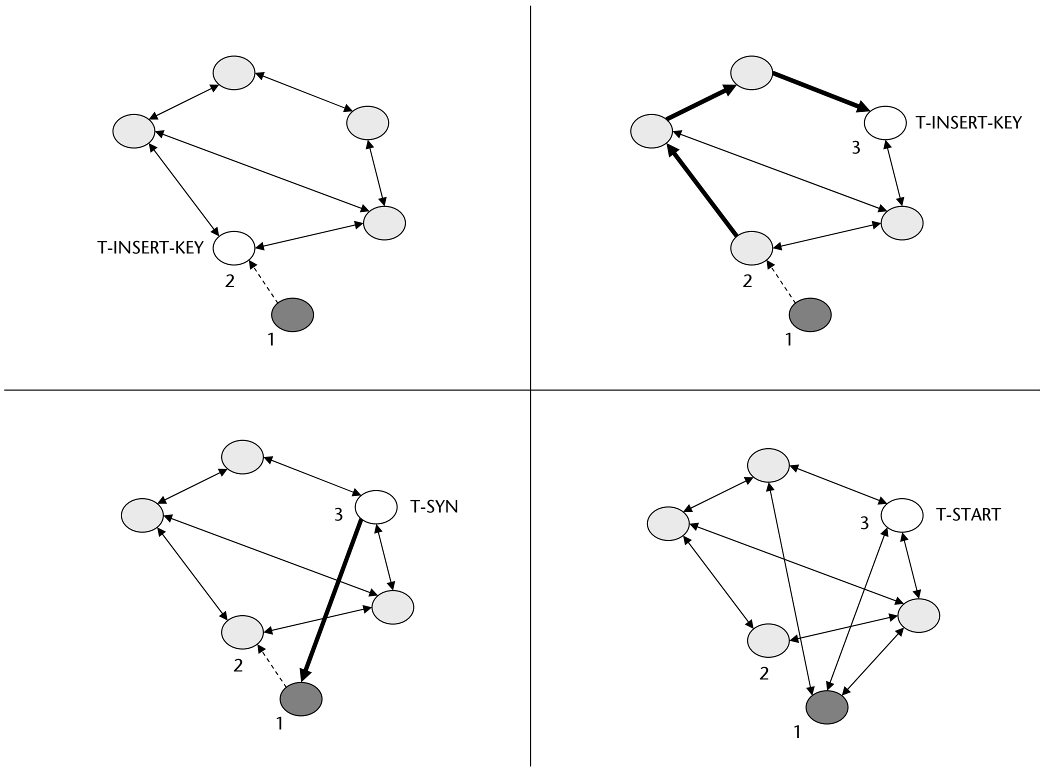

Figure 4.10: Joining and Neighbor Discovery: When a new node (Node 1) joins the network it first connects to a bootstrap node (Node 2) (upper left). The bootstrap node sends a T-INSERT-KEY message to the node that has the shortest Hamming distance to Node 1 (Node 3) (upper right). Node 3 then sets up a connection and connects to Node 1 via T-SYN (lower left). Finally, Node 1 receives a T-START packet and the routing table information about its new neighbors and can now communicate in STONe (lower right).

Routing Updates When STONe detects a broken link or a peer node, the node updates its routing table accordingly and sends a routing table update to its neighbors. The node's neighbors then update their own routing tables and send their updates to their neighbors. This process converges eventually and stops when no more updates occur. Additionally, nodes send their current routing tables to their neighbors in periodic intervals. We now analyze how many nodes' routing tables will be updated.

When the N addresses in STONe are randomly distributed, a specific node is expected to be in $S(N)$ routing tables throughout STONe. When a node leaves or joins STONe it takes at most $L(N)$ update messages for the information to arrive at all nodes with a total number of $S(N) \cdot L(N) = O(\log^2 N)$ messages in the worst case. However, the expected number of single updates is much smaller. The probability that a joining node has prefix length i with an arbitrary node in the network is

$$P_{PF}(i) = \begin{cases} \frac{1}{N_{max}}, & i = \log N_{max} \\ \frac{1}{2^{i+1}} & otherwise. \end{cases}$$

The number of nodes with prefix length i in the network is therefore $\frac{N_{max}}{2^{i+1}}$, and the expected value for the number of occupied prefix slots in the routing table is $E_{PF}(N) = O(\log N)$, since $E_{PF}(N) = \sum_{i=0}^{N_{max}} \left(1 - (1 - \frac{1}{2^{i+1}})^N\right)$.

The probability that a joining node has Hamming distance i is

$$P_{HD}(i) = \frac{1}{N_{max}} \binom{\log N_{max}}{i}.$$

The expected Hamming distance for a joining node is $\frac{\log N_{max}}{2}$, but because of the binomial distribution it takes about $(\frac{N_{max}}{2} - k)^{(\frac{N_{max}}{2} - k)}$ joining operations to find a node with Hamming distance k when $k < \frac{N_{max}}{2}$. Therefore, it is more likely that the STONe routing table finds a new prefix instead of a shorter Hamming distance. Given these probabilities, the number of update messages is $O(\log N)$ for the the original node, $O(\frac{1}{2^N} \log^2 N)$ for its neighbors, $O(\frac{1}{2^{2N}} \log^3 N)$ for their neighbors and so on. We get for the total number M of update messages

$$M = \sum_{i=0}^{N} \frac{1}{2^{iN}} \log^{i+1} N = O(\log N).$$

STONe clusters routing updates over time to reduce the number of messages in the network and avoid instabilities when the system converges. Whenever an event occurs, STONe starts a timer that expires after a short fixed time interval. All events of the node during that interval will be collected and broadcast to the neighbors. This is especially useful when several nodes join STONe at the same time and the system is unstable.

Figure 4.11 shows STONe's routing update messages and key exchanges compared to mix networks when a new node joins. STONe nodes require only $O(\log N)$ routing updates and key exchanges, whereas mix networks always require $O(N)$ which obstructs scalability of mix networks.

4.3 Random Routing

In addition to secure and scalable routing, STONe uses random routing to provide anonymity.

Algorithm 3 Send Fragment in Random Routing

```
send(p, dest)

p.im= NULL
p.dest= NULL
random_route(p, dest, last)
```

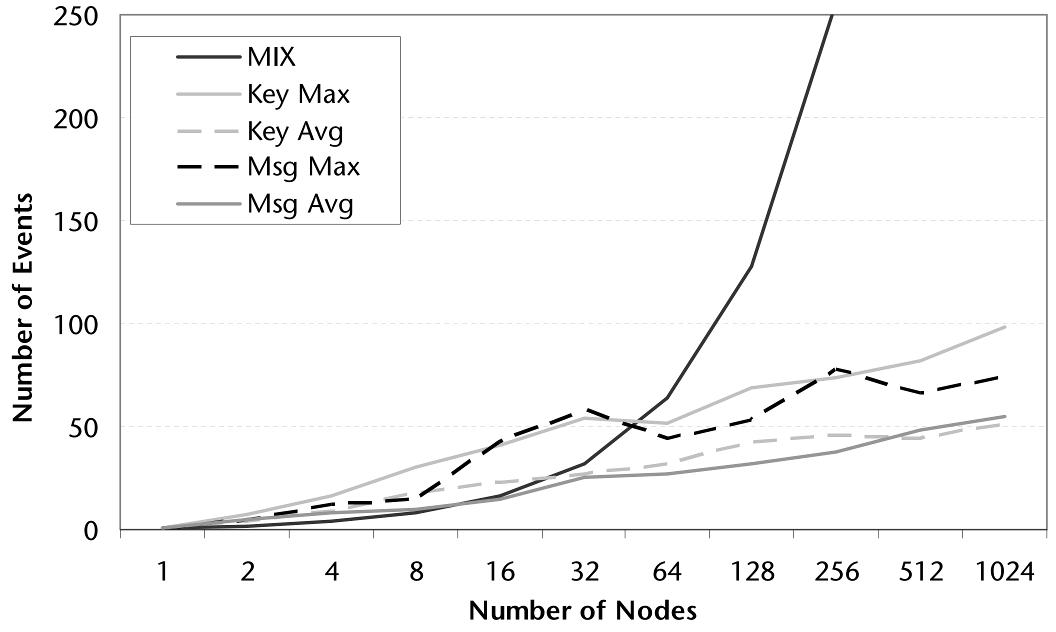

Figure 4.11: STONe under churn: The number of key exchanges and routing update messages in STONe increases logarithmically and is clearly smaller than for mix networks.

Figure 4.12 shows the scenario in random routing: Sender A forwards messages to receiver B and wants to hide the route. A picks an intermediate relay node I at random and forwards the packet to this node. The intermediate node I then forwards the packet to the receiver B. When B replies to the message from A she picks a different intermediate relay node J. Neither I nor J can carry out a Predecessor Attack to find S or R, respectively, because the probability that they are adjacent to A or B is small. The modified routing algorithm in random routing takes the following steps:

(i) Check whether the message is valid or not. If not, just drop it.

(ii) Check whether this node is the message's intermediate hop. If yes, copy the

Algorithm 4 Random Routing

```
random_route(p, dest, last)

if p.im = NULL then
    p.im= random()
    p.dest= dest
    route(p, p.im, last)
else
    p.im = NULL
    route(p, p.dest, last)
end if
```

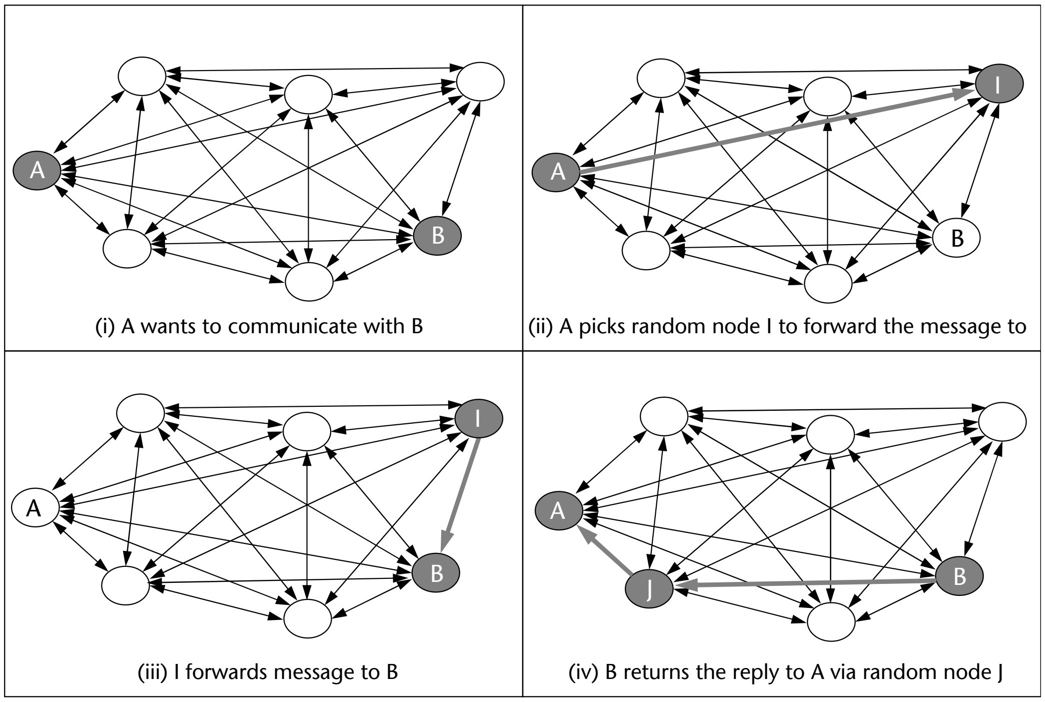
A
B
I
J
(i) A wants to communicate with B
(ii) A picks random node I to forward the message to
(iii) I forwards message to B
(iv) B returns the reply to A via random node J

Figure 4.12: Random Routing in a fully connected network: Node A communicates with node B through a random intermediate hop I.

final field from the message header into the destination field and forward the message towards the destination.

(iii) Check whether this node is the message's destination. If yes, deliver it to the application.

(iv) Forward the message towards the destination.

Random routing addresses both congestion, as well as traffic analysis. Congestion is less likely to occur, because potential bandwidth bottlenecks between a particular sender and receiver get automatically scattered across STONe. Because link utilization becomes uniform, certain traffic analysis attacks are more difficult. In addition, random routing reorders messages without additional buffering as in mixes, because every message takes a different random route through the network.

4.4 Synchronization in STONe

The remaining problem in Traffic Analysis with asynchronous random routing is *intersection attacks*. Any idle node in STONe that is not a sender or receiver forwards significantly fewer fragments than a sender or receiver does. Synchronization is the typical method for addressing this problem.

The main goal of synchronization in STONe is that links, and therefore nodes, should be utilized equally to make intersection attacks hard. The average rate in STONe, $\bar{\lambda}$, is the average per-link rate in the overlay, whereas links are simple point-to-point Internet connections. A high-bandwidth node has more virtual addresses and therefore more links. All links in STONe are utilized equally when every node sends at the rate $\bar{\lambda}$.

In general, synchronization can be global or local. In local synchronization traffic is synchronized only between two nodes. However, local synchronization, even among a limited set of nodes, would provide only local anonymity, even if it spreads globally [177]. This still leaves the door open for intersection attacks. This is similar to the end-to-end argument in systems design – local encryption does not necessarily provide end-to-end security [156]. In addition, these local schemes, even with extra traffic, often do not optimize for the bandwidth-delay product, since delay and extra traffic only depend on the local view of the mix. When the bandwidth-delay product grows, more data is in the pipeline between the sender and the receiver, and this requires larger retransmission buffers and more frequent transmissions to get optimal performance out of transport-layer protocols such as TCP.

A system low-level synchronization method would be hard to implement in STONe for several reasons: First, STONe is based on wide-area connections and TCP, and TCP's congestion control mechanisms may not always allow timely message delivery. Second, since STONe runs on the Internet with heterogeneous links and variable background traffic, available link capacities vary, and it would be hard to allocate fixed link capacities for synchronization. Random routing in contrast works on top of the existing Internet routing architecture and is able to balance traffic relative to the individual link capacities. Further, if STONe synchronizes traffic by padding all links with random cover traffic it wastes a lot of Internet bandwidth. In STONe's practical scenario, many people use it over DSL lines, and usually the link between the end system and the router is the bottleneck. STONe may have to share this link with other untrusted Internet applications. Fair sharing of network links between trusted and untrusted applications is required. Also, users may have to pay by traffic volume, and therefore they want to minimize traffic as well.

TCP already provides two end-to-end synchronization mechanisms: *congestion control* and *flow control*. Whenever a node sends a full window of data, the window increases incrementally. The current TCP window indicates the link capacity, and congestion control regulates the sender rate. Now, every node i starts to send data and tries to increase its send window size slowly to $\frac{1}{\lambda_i}$.

The synchronization algorithm presented in this section has the following idea: If for STONe node i's send rate λ_i, $\lambda_i < \bar{\lambda}$ the STONe node sends extra cover traffic to its immediate neighbors to reach $\bar{\lambda}$. The neighbors immediately drop any cover traffic packets. If a node cannot afford rate $\bar{\lambda}$ it automatically starts dropping packets because of TCP congestion control. This in turn reduces $\bar{\lambda}$. When other nodes see the drop in $\bar{\lambda}$ they stop increasing their send rates and restart later. If node i increases its send rate λ_i it is only allowed to increase it by an upper limit $\Delta\lambda_{max}$.

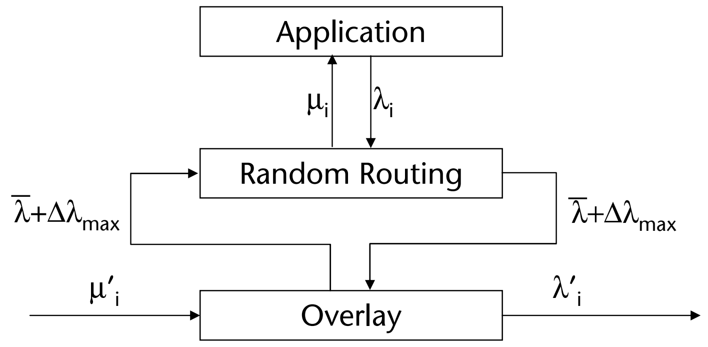

Figure 4.13: Traffic Rates on a STONe node: The actual network rates λ_i and μ_i in the synchronized overlay are larger than the average traffic rate $\bar{\lambda}$ in the STONe network. The individual application rates λ_i and μ_i can be larger or smaller than the average rate $\bar{\lambda}$.

Figure 4.13 shows the situation of different traffic streams in STONe without synchronization: λ_i' and μ_i' are the incoming and outgoing traffic rates in STONe's overlay network. $\bar{\lambda}$ is the average traffic rate across the network at time step t (when $t = 0$ we just leave it out):

$$\bar{\lambda}^t = \frac{1}{N}\sum_{i=0}^{N}\lambda_i^t$$

λ_i and μ_i are the local send and receive rates of applications on node i. For conducting the intersection attack, an adversary observes λ_i' and μ_i' across the network and correlates these values for traffic analysis. Both values include $\Delta\lambda_{max} \geq \bar{\lambda}$, which is the amount for possible extra cover traffic plus the traffic the STONe node forwards for random routing as an intermediate hop. Synchronization has to make sure that both values λ_i' and μ_i' are uncorrelated with the current traffic stream. λ_i' and μ_i' are always greater than $\bar{\lambda}$, whereas λ_i and μ_i are always less than $\bar{\lambda}$.

Before getting into the details of the synchronization algorithm, there are some important observations to make: First, in random routing, the sender forwards every message to a global random node, and therefore every node can measure $\bar{\lambda}$, the average traffic rate. Second, every STONe node is able to estimate the number of nodes N in STONe from the number of entries in its routing tables.

The algorithm is based on a credit/debit scheme to decide whether to send messages from the application, cover traffic messages or hold back in sending anything. This decision takes place whenever the application's send buffer is not empty, or the node has to forward a fragment in random routing (not proxy-based hop-to-hop forwarding).

Upon joining the network every node gets a minimum number of credit points (MIN_CREDIT) for each of which it is allowed to send one fragment from its send buffer. The node can acquire an additional credit point after it has forwarded N fragments (N is the number of nodes in STONe) in random routing.

When sending data, the node is always allowed to send a single fragment from the send buffer for every single credit point. Once a node runs out of credit it has to wait until it acquires new credit, ie. forwards more fragments, until it can send the next fragment from the send buffer.

When the send buffer is empty and there are already MIN_CREDIT credit points available, no more credit is added when the node forwards fragments. Instead, the nodes sends an extra fragment as cover traffic to a random neighbor whenever it has forwarded N fragments.

Algorithm 5 Synchronizing Traffic in a Domain of N nodes with time window T

```
1. Initialize
credit= MIN_CREDIT
fragment_count= 0

2. Forward Fragment in Random Routing
fragment_count++
if  fragment_count == N  then
  fragment_count= 0
  credit+ +
  Send Fragment
end if

3. Send Fragment
if  (send_buffer > 0 && credit > 0)  then
  credit–
  send_fragment()
else
  if  ( credit > MIN_CREDIT)  then
    credit- -
    send_cover_traffic()
  end if
end if
```

Algorithm 5 shows the final synchronization algorithm. When a node *receives a fragment as an intermediate hop in random routing* it counts the number of fragments to obtain the average rate in STONe. $\bar{\lambda}$ is the number of fragments forwarded over a time interval T. $\Delta\lambda_{max}$ is initially MIN_CREDIT over a time interval T and converges towards the number of fragments forwarded divided by the number of nodes N in STONe over some time interval T.

In random routing $\bar{\lambda}$ is the control parameter for global traffic synchronization. Given that random routing gets feedback on the current average traffic rate and is therefore *self-timing*, we can minimize additional cover traffic to make traffic patterns uniform across the network: Whenever a STONe node has received N fragments to forward in random routing it also sends one of its own fragments. Node i is allowed to increase λ_i beyond $\bar{\lambda}$ to $\bar{\lambda} + \Delta\lambda_{max}$ which will slowly increase $\bar{\lambda}$ and therefore every other node's cover traffic.

STONe has to configure the global parameter $\Delta\lambda_{max}$, which is a tradeoff in the bandwidth-delay product of the whole system. An individual node with average latency d_i between its neighbors has a bandwidth-delay product $\Pi(i)$ of

$$\Pi(i) = \lambda'_i \left(d_i + \max\left(0, \frac{1}{\bar{\lambda}} - \frac{1}{\lambda_i}\right)\right) = \left(\bar{\lambda} + \Delta\lambda_{max}\right)\left(d_i + \max\left(0, \frac{1}{\bar{\lambda}} - \frac{1}{\lambda_i}\right)\right)$$

When node i is idle (i.e. $\lambda_i = 0$) or busy (i.e. $\lambda_i = \bar{\lambda}$) we get

$$\Pi_{idle}(i) = \Pi_{busy}(i) = d_i(\bar{\lambda} + \Delta\lambda_{max})$$

The goal in STONe is to avoid large oscillations of $\bar{\lambda}$, since this supports intersection attacks. Therefore, to ensure fairness all λ_i should be distributed uniformly. It takes N messages on average for all STONe nodes to notice that $\bar{\lambda}$ has increased. And ideally, the bandwidth-delay product $\Pi(i)$ is as small as possible. This exact outcome happens when $\lambda_i - \bar{\lambda}$ is minimal. This is exactly the case when $\Delta\lambda_{max} = \frac{\bar{\lambda}}{N}$, which is the expected standard deviation of λ when λ_i are uniformly distributed. After k sequential steps (ie. forwarded fragments) with $\bar{\lambda}^t = \frac{(N+1)\bar{\lambda}^{t-1}}{N}$ we get

$$\bar{\lambda}^k = \left(1 + \frac{1}{N}\right)^k \bar{\lambda}$$

This has a shorter response time to changes, and causes cover traffic to be linear in the number of nodes N. If a node has to send a burst of data at rate λ_{max} it takes

$$k = \frac{\log(\frac{\lambda_{max}}{\bar{\lambda}})}{\log\left(1 + \frac{1}{N}\right)}$$

steps until it reaches λ_{max}.

The total required bandwidth per node is $(2 + \frac{1}{N})\bar{\lambda}$. In addition to the average send rate for every node $\bar{\lambda}$ the maximum increase is $\frac{1}{N}\bar{\lambda}$. When a bottleneck node has bandwidth b then $\bar{\lambda} \leq \frac{N}{2N+1} b \approx \frac{1}{2} b$ for large N. The system is secure against intersection attacks, since all nodes adapt their send rate to $\frac{N+1}{N}\bar{\lambda}$ synchronously, even though communication is still asynchronous.

Session Scheduling An extension for future work is to synchronize sessions on application-level. This does not work for highly interactive applications such as web browsing and instant messaging, but it does work for file transfers. A distributed scheduler caches requests for network tasks until it has collected a large enough number that overlaps. It then starts these tasks at the same time.

4.5 Anonymous Communication

An anonymous application layer poses many challenges. First of all, an application that communicates through this application layer can potentially be untrusted outside the TCB and should not be able to link an application endpoint to a real IP address. It is also crucial to have an anonymous name service running in the network, since a query to an external name server such as DNS would be a weak spot in the system that is easy to attack.

STONe integrates end-to-end anonymity with anonymous self-certifying credentials by providing two services to applications: STONe Sockets and Trusted Name Service (TNS). Using these two building blocks it is straightforward to turn an existing socket application into an application that runs on STONe and is robust against traffic analysis.

4.5.1 STONe Interface

The most important part of Anonymous Communication is the interface between the network and the application. STONe already provides a secure infrastructure that is robust against traffic analysis. Yet, an adversary who receives STONe fragments legitimately at the communication endpoint would still be able to see the STONe addresses in the clear. Hiding the STONe address not only protects against address filtering but also gives the adversary valuable information about STONe's topology. The adversary is then able to obtain secrets about the randomized hypercube structure – a main support for protection against traffic analysis.

STONe Capabilities hide the original STONe address from the application. These capabilities are opaque random strings and have the sole purpose of providing a handle for the application to communicate with the peer. Only the STONe router is able to decrypt these capabilities to obtain the original STONe address. A 128-bit capability for destination d contains a 64-bit STONe address T_d and a 32-bit STONe service ID S. STONe obtains the capability by encrypting the T_d and S under the STONe session key for session j on node i SK_{TC_i}, which is derived from the secret Trusted Computing key SK_{TC_i} and a 32-bit nonce n_t:

$$C^S_{T_d} = F_i(T_d \mid S) = E_{K_{TC_i}}(T_d \mid S \mid n_t)$$

STONe uses per-session keys to convert capabilities into STONe addresses and vice versa. Otherwise the adversary is able to correlate capabilities among different applications or different sessions of the same application to determine which applications communicate with the same destination. The nonce n_t belongs to the socket state. Every time the application reconnects to the same node and opens a new session, the STONe router increases t by 1 and generates a new n_t by getting a new random number r_j.

STONe provides two services to applications that are built around these capabilities: STONe Sockets and the Trusted Name Service (TNS). With these two building blocks it is straightforward to turn an existing socket application into a STONe Socket application that runs on STONe and is protected from attacks on anonymity. Additionally, STONe provides TSOCKETS – a TCP proxy that translates STONe connections into TCP connections. Any TCP socket application can use TSOCKETS to connect to STONe. The 32-bit IP addresses that are visible to the applications are opaque and randomized, making them similar to STONe capabilities.

STONe Sockets One application building block of STONe is the STONe Socket interface. STONe Sockets are anonymous application-level endpoints. Externally, they provide fully randomized IP addresses over TCP and UDP, but internally, STONe Sockets use a Trusted Datagram Service (TDS) and a Trusted Stream Service (TSS). STONe generates random IP addresses from STONe's network addresses– STONe Capabilities. *STONe Capabilities* look like opaque random strings, but for the STONe Router they contain meaningful session information. In every per-session capability there is encrypted a 64-bit STONe address T_d and a 32-bit STONe service ID S_d pointing to a destination d in STONe:

$$C_d = E_{K_{TC_i}}(T_d \mid S_d \mid n_t).$$

The first 64 bit of the capability contain the 32-bit IP address and a 32-bit port number. When an application opens a new session STONe generates a new random nonce n_t associated with this session and computes a new capability C_d. Since K_{TC_i} is hidden from the application, only the TCB is able to encrypt and decrypt capabilities.

Figure 4.14 shows how STONe converts capabilities into STONe addresses and service IDs, and the other way around. The application places capabilities into the STONe Socket calls, and the STONe Socket library passes the capabilities to the STONe router within the TCB for decryption and further use. In the converse, when a STONe Socket call returns a capability to the application, the STONe router encrypts the

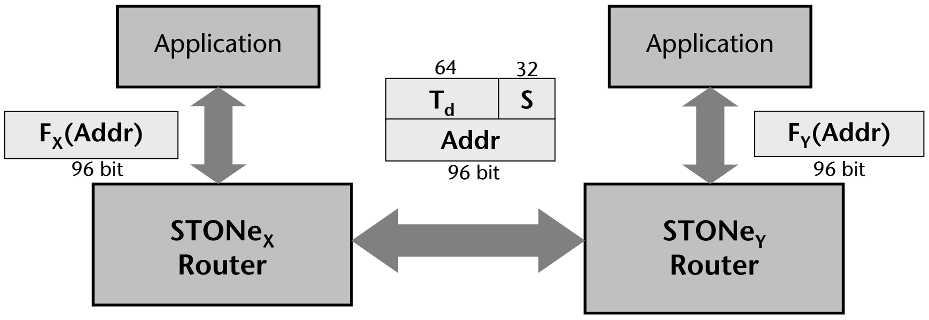

Figure 4.14: Conversion of STONe capabilities: An application on router X receives the capability encrypted under F_X that depends on the built-in Trusted Computing key. Between the two routers the capability gets transmitted in plaintext, but the message is encrypted as a whole.

STONe address and service ID and passes the resulting capability to the application. The encryption function depends on the built-in AES keys K_{TC_X} and K_{TC_Y}, such that $F_i(A) = E_{K_{TC_i}}(A \mid r_j)$. All STONe has to cache is the random per-session nonce r_j for every session.

Trusted Datagram Service – TDS

STONe offers TDS datagram service for connectionless communication. A STONe datagram packet can consist of multiple STONe fragments with fixed-size unit for network communication. Upon receipt, STONe reassembles multiple fragments together to one STONe datagram packet. The datagram service is unreliable and unacknowledged, and the application is responsible for resending packets. TDS is an adaptation of UDP that uses STONe's capabilities.

Trusted Stream Service – TSS

STONe also offers a connection-oriented stream service called the Trusted Stream Service (TSS). STONe cannot use TCP for end-to-end communication for multiple reasons: First, point-to-point links in STONe already use TCP, and in case of a congestion on the link the protocol increases time-outs, causing the upper level TCP also to increase timeouts and queue packets for retransmission. Then both TCPs try to retransmit packets, which only makes the problem worse. Eventually, they will have to give up and reset the connection. Second, packet reordering caused by random routing is a problem in TCP. TCP's fast retransmit algorithm interprets excessive packet reordering as a loss and tries to retransmit packet, thereby wasting valuable bandwidth [178]. Retransmission is not the only problem, and TCP does not detect these losses as reorderings but as buffer overflows. Once it detects any losses, it starts congestion control mechanisms to regulate throughput [46, 140].

TSS avoids these problems altogether, since it does not have congestion control and is also aware of packet reorderings. TSS is similar to TCP except for flow control and congestion control, but its protocol states are equivalent. STONe already uses random routing and synchrony as an efficient measure against congestion. Initially, TSS does a three-way handshake to establish the connection and initialize the buffers for flow-control. A STONe stream consists of STONe fragments. All nodes in the STONe network use the same retransmission timeout to protect against attacks that attempt to identify a communication channel based on their retransmission patterns. The retransmission timeout depends on the maximum path latency in the overlay, which is $log(N)$-times the maximum per-hop latency, where N is the number of nodes in the overlay.

Trusted Name Service – TNS

Because opaque capabilities are only application-specific handles within socket applications, STONe requires a global naming infrastructure that maps anonymous names to capabilities, similar to what a pseudonym server does [123]. TNS is a trusted application within STONe, and therefore common attacks are hard to mount from any malicious application. TNS is different from a DNS server because is it easy to register new entries. Further, TNS can handle different application-level services for a single node as well, such as a shared address book. TNS is similar to Tor's Location-Hidden Services, where two parties use a third random node as a rendez-vous point for exchanging services [75].

An application registers names along with an application-specific public key PK_{TNS_d}. TNS is self-certifying because each TNS request returns an entry in which the capability is XORed with the hash of the public key. Only when the application has the entry's public key it is able to obtain and use the capability.

TNS uses a challenge-response mechanism during name registration to ensure that the mapping between the public key and the network address is correct. This certification mechanism prevents falsifying of identities, while allowing anonymous communication. When a node joins the system, its trusted infrastructure learns about the addresses of the name servers, but these addresses stay invisible to the applications.

An application registers a name for a service at the name server using the `tns_register` function in the STONe Socket library. If an application wants to register a service it needs to provide PK_{TNS_d} for it. TNS executes a challenge-response mechanism to check that the application really owns the private key:

Application $\rightarrow$ TNS:	$< ID^s_{TNS_d},\ PK^s_{TNS_d},\ C^j_i >$
TNS $\rightarrow$ Application:	$<r>$
Application $\rightarrow$ TNS:	$<x := E_{SK}(r)>$ with $SK = SK^s_{TNS_d}$
TNS:	verify $D_{PK}(x) = r$ with $PK = PK^s_{TNS_d}$

First the application calls `tns_register`, causing the STONe Socket to send to TNS the capability C_i^j, as well as the name $ID_{TNS_d}^s$ and PK_{TNS_d} of the service the application wants to register. TNS generates a random string r and sends it as a challenge to the application. The application encrypts r using the private key SK_{STONe_i} and sends the ciphertext x back to the TNS. TNS verifies that the decrypted ciphertext x is equal to r. After these operations the TNS server is sure that the capability x belongs to PK_{TNS_d}, and it can register $<ID_{TNS_d}, PK_{TNS_d}>$ as a certified name.

When a client application wants to look up a capability, it queries TNS using the $<ID_{TNS_d}, PK_{TNS_d}>$ tuple or just ID_{TNS_d}. An application resolves a name in TNS and obtains the corresponding capability through the `tns_query` function in the STONe Socket library. It can search for any name and get back a list of corresponding capabilities. If TNS provides a public key the query is certified, and the application can trust that the capability belongs to the public key:

Application → TNS:	$<ID_{TNS_d}^s, PK_{TNS_d}^s>$
TNS → Application:	$<cap>$

The application then retrieves the original capability C_i^j by XORing cap with PK_{TNS_d}. The key for every entry is the name and public key. It is also possible to register the same name and the same public key multiple times if the same entity is located at multiple destinations. TNS returns a list of all entries. TNS also allows wild-card searches without passing the public key to the server when the corresponding flag is set during the register operation. This is a privacy option. Registering the same entry at multiple locations does not violate anonymity because the capabilities do not provide any locality information.

Normally, such a server for secure and anonymous peer-to-peer names would have to store N^2 different names for one service in a network of N nodes, because every node would have to have a different address. However, trusted computing is able to map multiple network addresses to a single one, and therefore every node in the network needs to store only one entry.

TNS is located locally on STONe nodes, and every node learns from its neighbors the location of the closest TNS node. However, when a TNS node crashes, all the information is lost. Without TNS it is not possible to obtain capabilities for communication in STONe. Therefore, TNS must be highly available and implemented as a distributed name service where data gets replicated dynamically.

Trusted SOCKS Proxy – TSOCKS

Some applications do not require the full strength of 96-bit capabilities. Rather backwards compatibility to the existing Internet is desirable. For this purpose TSOCKS provides an proxy server between TSS and TCP that is similar to SOCKS [119]. SOCKS is a TCP proxy that allows multiple clients to share the same outgoing connection on a proxy, thus reducing the number of ports a firewall has to allocate. For example, Tor uses SOCKS to connect to Privoxy to eliminate any application-level privacy-related information in web browsing [75, 18]. The purpose of TSOCKS, however, is to generate random IP addresses for applications to communicate through STONe. STONe's 96-bit opaque capability translates to a 32-bit IP address and a 16-bit port number, thus reducing the effective address length 48-bit. Because of the address length limitations, applications running on top of TSOCKS do not have the same guarantees against denial-of-service attacks as STONe socket applications. Furthermore, the connection between the TSOCKS proxy and the client needs to be sufficiently protected against an eavesdropping adversary. Ideally, the TSOCKS proxy is a trusted process on the local platform.

4.5.2 STONe Applications

There are several examples of applications in distributed systems that strongly benefit from STONe's security and anonymity.

Trusted Load-Balancing – TLB

STONe's enhanced security provides simple load-balancing without DHTs for server applications. The server registers several entries with $ID_{TNS_i} = <App, Loc_i>$, where Loc_i is the number of the server, and all servers have to be enumerated consecutively from 0 to $Loc_{max} - 1$. A separate key server carries out admission control by randomly distributing public keys to all clients. In this process the application belonging to the same capability will only see one of the Loc_{max} public keys. The application uses that public key to request the corresponding destination capability from TNS. To minimize key distribution overhead, all server instances of App have the same public key PK_{TNS_i}. Giving all instances the same PK_{TNS_i} does not compromise security, because they all relate to the same server application. TLB is robust against attacks on STONe – especially traffic analysis attacks – which protects load-balancing in a global network as strongly as if it took place in a local server cluster behind a firewall. TLB is different from DNS-based load-balancing because TNS is trusted and is hard to sabotage.

Application → Key Server:	query key
Key Server → Application:	$PK^s_{TNS_d}$, Loc_i
Application → TNS:	$<<App, Loc_i>, PK^s_{TNS_d}>$
TNS → Application:	C^s_d

Figure 4.15: Trusted Load-Balancing Protocol

Anonymous File System Anonymity and group-based certification without a central administrator are important issues for global file systems (e.g.[102]). Assume two parties – Alice and Bob – want to share a file: First, the global file system should be able to store Alice's file in multiple locations, because nodes that contain the file may enter and leave the network arbitrarily. Second, a third untrusted party should not be able to learn that Alice is the owner of this file. Third, any trusted party should be able to verify that she got the right file from Alice and not a fake one. Our Anonymous File System on top of STONe has these properties. Every file in the system has its name and the owner public key stored on TNS. TNS maps a file to multiple network addresses. On a node every file gets a unique service ID. TNS models the file system in this fashion; when Alice wants to store her file in the file system, she registers it with TNS. Then, Bob looks up Alice's file by querying TNS. Content-distribution systems such as Freenet [62] and BitTorrent [5] or other hierarchical file systems could be built on top of this anonymous filesystem.

Application $\rightarrow$ **TNS**:	$< File - /Directoryname,\ PK^s_{TNS_d} >$
TNS $\rightarrow$ **Application**:	C^s_d

Figure 4.16: Anonymous File System Protocol

Anonymous Instant Messaging Instant Messaging is usually hard to anonymize because the parties participating in the conversation have to be logged onto the system. The provider of the service can easily observe the communication channels at any time, which parties participate and where they are logged on. Furthermore, an attacker with access to the links can easily detect conversational traffic patterns within a session. We built a prototype of Anonymous Instant Messaging using STONe. Alice logs onto the system by registering her pseudonym and her public key on TNS. If the other party Bob wants to talk to Alice, he looks up her pseudonym on TNS, verifies that this is her public key (if she wants to reveal her identity to him), and starts to talk anonymously to her over the system. Bob obtains the public key beforehand through a different trusted channel similar to a web of trust (e.g [86]). When Bob initiates the conversation, Alice verifies Bob's public key to make sure that he is the right person. In Anonymous Instant Messaging, only Alice and Bob know that they are talking with each other.

Application $\rightarrow$ **TNS**:	$< IM\ Pseudonym,\ PK^s_{TNS_d} >$
TNS $\rightarrow$ **Application**:	C^s_d

Figure 4.17: Anonymous Instant Messaging Protocol

4.6 Compromised Trusted Computing Hardware

So far we have excluded the hardware attack on the Trusted Computing system, since it is hard to do and can only be done at a node that is physically available. Because STONe's address depends on a built-in key, it is not possible to forge other Trusted Computing nodes in STONe after extracting the key from the TCB because STONe will detect duplicates. Furthermore, an adversary is unable to get access to the system just by knowing the built-in keys. She has to obtain the STONe secret S_{STONe} to successfully join the network and also to decrypt messages.

When an adversary learns the secret keys she is able to emulate a full TCB and join the network. However, having these trusted keys she can mimic only one TCB. When this happens and remains undetected the following scenario takes place: The adversary is able to monitor all traffic that goes through this node, and because of random routing this can be any sender/receiver pair. Furthermore, she is able to learn the IP addresses of the local routing table and can assemble a membership list of the network.

It is therefore crucial that hardware compromises are detected immediately, so that compromised nodes are not able to join the network through remote attestation. In addition to key-based attestation the TCB could use other platform authenticity tests, such as Remote Physical Device Fingerprinting [114] to ensure that the TCB runs on the actual hardware. It is also important to verify that the TCB does not contain any backdoors.

Chapter 5

STONe Implementation

Normally, STONe would be considered a part of the internal kernel network stack that provides a different new transport layer. This is different from Tor's approach, which uses a network of proxies. Unfortunately, our experimental testbed, PlanetLab, does not support kernel extensions in the virtualization layer, and therefore we have to implement STONe as user-level processes. Kernel processes typically get higher priority, and they do not require expensive copying of buffer data between kernel- and user-space.

5.1 Trusted Computing Base

5.1.1 Required Hardware

Today there exist several approaches for Trusted Computing hardware. The Trusted Platform Manager (TPM) contains primitives for remote attestation and sealed storage [21]. These hardware chips are already built into most PCs. Windows and Linux provide device drivers for them. TPM also provides a trusted boot mechanism that protects against local attacks.

In addition, STONe requires strong process isolation against attacks from the OS kernel. Intel is planning to ship LaGrande in future PC platforms [99], and AMD's equivalent product Pacifica provides the same functionality on future AMD platforms [28]. Microsoft is going to provide support operating systems support for strong process isolation in their NGSCB architecture [79].

5.1.2 TCB Software Emulation

We build STONe on top of Linux, but unfortunately, Linux does not have support for a fully functional TCB, and most devices do not have Trusted Computing hardware, either. The choices are either to implement a fully-functional device driver that emulates a TPM and run a trusted operating system on top, or we assume the commodity operating system to be the root of trust and emulate TCB on top. In previous approaches virtual machines implemented Trusted Computing such that every trusted process is mapped to one virtual machine [87]. The virtual machine monitor is the root of trust and provides the interface to the trusted computing system. In our TCB emulation we pick a similar approach and use Linux to be the root of trust, since we want to fit it in the architecture of PlanetLab.

The TCB emulator needs to provide cryptographic functions, such as signature scheme, hash function and random number generator, as well as attestation and operating system support. We had to make a design choice for the remote attestation protocol. When using the standard attestation protocol the TCB public key gets revealed to the verifying entity. This is not a problem in local attestation, since the verifying entity is the same as the signing entity – the local TCB. However, it is a problem in remote attestation, since signing and verification happen on different platforms, and therefore remote attestation uses group signature schemes that preserve this privacy of the TCB. Otherwise, a peer could tell that a specific entity signed the message. In short summary, Figure 5.1 shows the TCB inteface that is required to run STONe:

`tcb_hash()`
: This computes the hash of the value `in`. The current implementation uses `SHA1` from the OpenSSL crypto library with a built-in TCB hashing key that is the same on all TCBs.

`tcb_srand()` and `tcb_rand()`
: `tcb_srand` initializes the Random Number Generator with start value `seed`, and `tcb_rand` computes the next random number in the sequence. Both functions map to `RAND_seed` and `RAND_bytes` from the OpenSSL crypto library respectively.

`tcb_seal()` and `tcb_unseal()`
: `tcb_seal` encrypts plaintexts using the secret built-in platform key, and `tcb_unseal` decrypts ciphertexts using the secret built-in platform key.

`tcb_pk_sign()` and `tcb_pk_verify()`
: This pair does public key signing and verifying using the built-in public key. These functions map to `RSA_sign` and `RSA_verify` from the OpenSSL crypto library.

`tcb_at_sign()` and `tcb_at_verify()`
: Invokes the attestation protocol on a given binary. If `fd` points to a remotely connected socket, the primitive invokes remote attestation. Otherwise it uses local attestation within the TCB. Normally, remote attestation should use DAA [51] or

```
typedef struct msg_s {
     int m;
     int len;
} msg_t;

typedef struct bin_s {
     void *p;
     int len;
} bin_t;

typedef struct sig_s {
     void *s;
     int len;
} sig_t;

int tcb_hash(int in);
int tcb_srand(unsigned int seed);
int tcb_rand();
int tcb_seal(msg_t msg);
int tcb_unseal(msg_t *msg);
int tcb_pk_sign(msg_t msg, sig_t *sig);
int tcb_pk_verify(msg_t msg, sig_t sig);
int tcb_at_sign(int fd, bin_t bin, sig_t *sig);
int tcb_at_verify(int fd, bin_t bin, sig_t sig);
int tcb_at_join(int fd);
```

Figure 5.1: TCB Interface for STONe

a similar group signature scheme, but in this emulation we currently use standard public key signing methods to simplify the complexity of the TCB emulation. When a DAA implementation is available this can be easily replaced by a call to the group signature scheme.

`tcb_at_join()`
: Joins a group signature scheme for remote attestation and obtains the shared signing key. This function only works for remote attestation.

Trusted Operating System Support   In addition to these primitives, Trusted Computing requires support from the operating system. In our TCB emulation there are three issues: (i) Trusted Computing requires strong process isolation, for which virtual memory protection is not sufficient. (ii) There must be a loader for every `exec()` call that attests the binary locally. (iii) The operating system needs to provide an interface for remote attestation.

Strong Process Isolation

We achieve strong process isolation by using the Linux VServer virtual machine implementation [11]. PlanetLab [38] is a common testbed for distributed applications and uses VServer to separate different slices from each other. Since we do not have control over the VServer in PlanetLab, the Linux kernel in the virtual machine itself is the root of trust. Because Linux itself does not provide strong process isolation, there can be only one trusted process or multiple untrusted processes per kernel. We create a slice on PlanetLab for every different application and achieve the desired strong process isolation.

Secure I/O

The only secure I/O device that we need to emulate for Trusted Overlay Networks is the network adapter. We emulate secure network I/O by encrypting data from the application to the peer application after the key exchange is done, as shown in Figure 5.2.

Local Attestation

The operating system needs to provide local attestation to establish a chain of trust between the application and the trusted hardware during secure boot. First, the TCB attests the OS Loader, which is usually the BIOS of a PC, and starts it. The OS Loader then attests the OS and loads it. Finally, the OS attests any trusted application and executes it. The TCB emulation provides a wrapper function for the `exec()` system call that executes the local attestation protocol.

Remote Attestation

Figure 5.2 shows how two trusted software stacks attest applications to each other using remote attestation. First, the applications conduct a standard TCP handshake. After the TCP connection has been established, the application does a key exchange and sets up an encrypted communication channel. Establishing the encrypted communication channel before attestation takes place is necessary to protect against replay attacks. Then the connecting TCB joins the group signature scheme using the `tcb_at_join` command. It starts the remote attestation protocol over that connection using the `tcb_at_sign`. The TCB emulates the remote attestation protocol, but the application handles all errors. Theoretically, we could use IPsec or TLS/SSL to set up a secure network or socket layer, but we do not require all the functionality on this level, and as discussed earlier it is an important principle in STONe to integrate handling of security errors and application errors on the higher level. Encrypting data between the nodes on this level is completely sufficient, and STONe handles integrity of the communication on the higher levels.

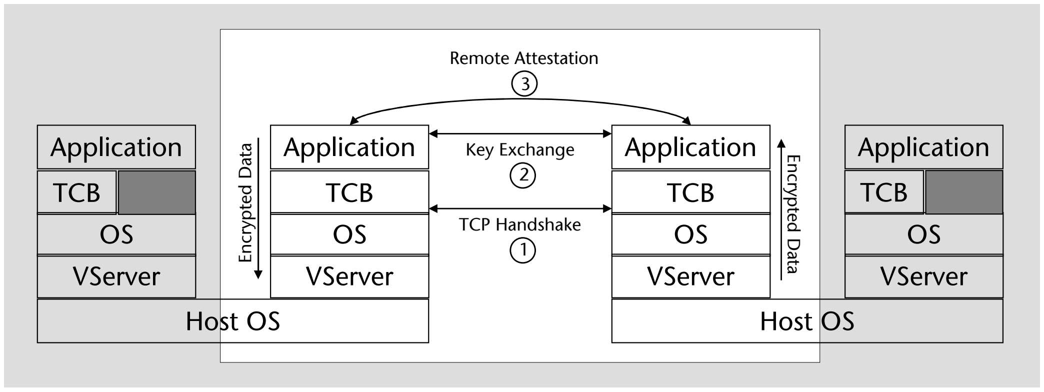

Figure 5.2: Emulating Remote Attestation and Secure Network I/O on PlanetLab: STONe emulates the TCB on top of the Linux OS in the VServer. Every VServer slice automatically provides strong process isolation. The application contains Trusted Overlay Networks and the actual application.

5.2 Implementing Trusted Overlay Networks

With the TCB emulation on PlanetLab, we build Trusted Overlay Networks (TON) – the base of STONe. Here we outline only the general implementation issues, but in STONe we chose to implement Trusted Overlay Networks as socket proxies, since they give the maximum amount of flexibility for any type of application.

Figure 5.3 shows the implementation of Trusted Overlay Networks: Every node has two PlanetLab slices – one that runs the TON software, and another that runs the application. The application communicates through TON with its peer nodes and can be trusted or untrusted.

The join protocol is simple: A node that wants to join TON first conducts a TCP handshake with another node in TON. In the next step the two nodes do a key exchange. Finally, they complete remote attestation.

In the current setting TON has to be implemented in user space because PlanetLab's current configuration does not support kernel modules. As we will see this incurs some performance penalties, but we will also obtain an estimate of how STONe would behave in kernel space.

So far we have explained the details of TON and outlined how to build TON on PlanetLab. In the next sections we explain the design and implementation of STONe, a Trusted Overlay Network that provides secure, reliable, and anonymous communication.

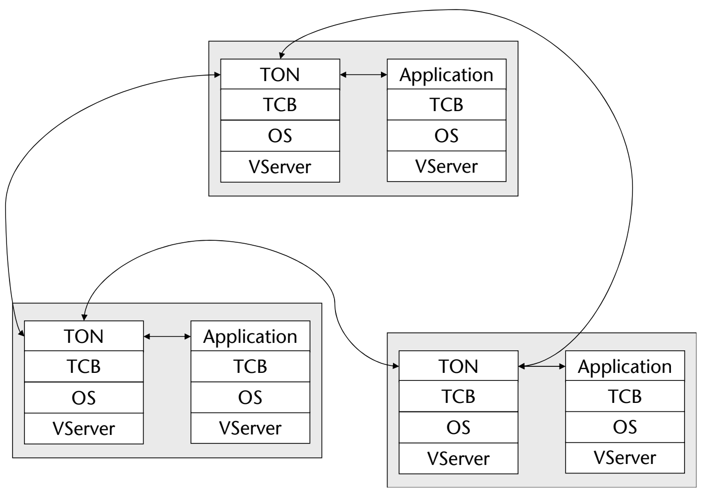

Figure 5.3: Implementing Trusted Overlay Networks: This figure displays the Trusted Computing stack on every platform. Trusted Overlay Networks (TON) sit on top of the TCB and connect other TON nodes.

5.3 STONe Implementation

We implemented a prototype of STONe under PlanetLab. The STONe Proxy and STONe Router in the system architecture are different processes, and they communicate via pipes and sockets. The STONe Socket library provides an API for applications to connect to the STONe Router. TNS is implemented as an application that runs in the TCB.

STONe Proxy The STONe Proxy decrypts incoming and encrypts outgoing fragments and communicates with the router. The most important decision in the implementation of the STONe Proxy is the choice of UDP vs. TCP. The STONe proxy uses TCP for a few reasons:

First, nodes may leave and enter STONe frequently. TCP connections can automatically detect when a node leaves STONe because the peer's TCP/IP stack closes the connection through explicit TCP FINs – even when the router application crashes. Second, TCP ensures ordered message delivery, allowing the traffic to be encrypted using a simple stream cipher. Third, TCP flow-control provides reliable transmission even over lossy paths, and TCP congestion control allows nodes to use up all the available bandwidth along the path. Fourth, TCP makes it easier to set up incoming connections to nodes within NATs in peer-to-peer networks. Typically, it is not possible for a node behind a NAT to receive any connections without additional support. In STONe, the node behind the NAT connects to another node using TCP, and when the connection goes this way the NAT often forwards the port automatically. In UDP only the node behind the NAT can send nodes outside, but it cannot receive any packets. After connection setup in TCP both nodes can send messages back and forth. For an overview of all possible solutions to this "Hole Punching" in TCP and UDP, see [83]. And fifth, TCP automatically buffers messages and slows down connections through congestion. When the STONe Proxy cannot forward any messages and does not pick up arriving ones, TCP automatically slows down the sender rate of the adjacent node. Therefore, no packets have to get dropped in the STONe Proxy. Finally, TCP congestion control synchronizes traffic in anonymous routing.

In addition to packet relay on STONe's overlay links, the STONe Proxy also encrypts data. However, it does not use SSL/TLS or IPsec for several reasons: First, authentication in STONe is done by remote attestation, and these protocols use their own authentication mechanisms, which have to be adapted for Trusted Computing. Second, content in STONe application data is already signed by the application, and STONe only requires a checksum for STONe's packet header to protect against tampering. SSL/TLS would impose additional overhead to signing and verifying messages. Third, IPsec has difficulties with NATs and firewalls, and we want to make STONe as transparent as possible to maximize the size of the anonymity set.

STONe Router The STONe Router is a process different from the STONe Proxy to provide resilience in case of an attack. When the STONe Proxy gets overloaded or compromised for some reason the Router spawns a new Proxy. After reconnecting to its neighbors, it then continues operation without much interruption. Further, this separation enables parallel processing of cryptographic operations and message forwarding on modern microprocessors that have multiple cores. Pipes under Linux provide the basic mechanism for interprocess communication between the Router and the Proxy. The Router does the initial handshake and then hands off the file descriptor of the TCP connection to the Proxy via access control messages on the internal pipes.

Interprocess Communication Pipes in STONe's interprocess communication are unidirectional, and we use two pipes to establish bidirectional communication. Initially there is a socket pair between the Router and the STONe Proxy to signal events and pass along file descriptors. Whenever a new node connects to the system, the Router sends a signal to the STONe Proxy to set up the new connection. For every connection the Router adds a new bidirectional pipe. Pipes limit the maximum possible STONe fragment size to 4kB since this is the current maximum buffer limit under Linux. When the buffer is full the Router gets blocked and has to wait for the Proxy to continue. There are still possibilities to optimize STONe by adding more buffers, but this is beyond the scope of this work and is left for future research.

Figure 5.4 shows STONe's interprocess communication with different applications. The STONe Router communicates with every single application through the Router

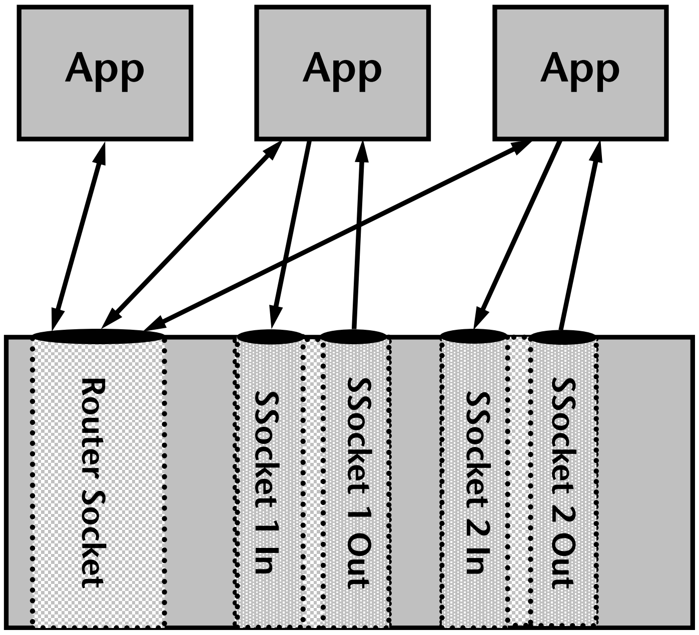

Figure 5.4: Interprocess Communication in STONe: The layout for interprocess communication in STONe consists of applications and the Router. Applications send commands to the Router and receive replies through the Router socket. For every STONe address the application binds it gets two unidirectional pipes.

socket, which is a standard Unix or Inet socket. The Router socket takes commands for the STONe network stack from the application:

`SK_CAP`
: Given a service ID the STONe Router computes the opaque capability for the local host and returns it to the application.

`SK_BIND`
: Binds a STONe Socket to the a local STONe address. The bind operation generates two unidirectional pipes between the application process and the STONe Router process.

`SK_CONNECT`
: Connects the STONe Socket to another listening STONe Socket. The connection uses the TSS protocol.

`SK_LISTEN`
: Turns the STONe Socket into listening state for incoming connections.

When an application binds a new address to a STONe Socket the STONe Router generates a new pair of unidirectional pipes for this STONe Socket. An application sending data through the STONe Socket Library writes the data into the pipe, and the STONe

Router then forwards it to the destination. If the STONe Socket is connected via TSS, this is stream data. If the STONe Socket is not connected and uses TDS, the STONe Socket Library copies the data into a single STONe packet and writes it into the pipe. This packet contains the destination capability in the header.

Process Synchronization The STONe Proxy and Router both use asynchronous I/O and an event-based programming model. Both have a command queue and data queues for every TCP connection to adjacent nodes. When a new node connects, the router sends a message to the Proxy, and the Proxy sets up a new data queue for this node. When an adjacent node leaves the network the Proxy sends a message to the router. It is important that the Router and Proxy are in sync when a node connects or disconnects, since otherwise data may be lost.

We circumvent deadlock problems by having only one synchronization point. The Proxy waits only when no packet arrives from the network and no data arrives from the Router. The Router waits only when there is no socket data, and when there is no data from the proxy as well. The main deadlock problem is the internal data pipe, whose size is only 4kB. STONe needs to make sure that both proxy and router are not trying to write on a full buffer at the same time. We solve the problem by blocking the Router, such that no data gets lost. Further, we use the TCP buffer in the proxy for buffering STONe packets: Whenever the proxy encounters a full pipe it does not receive any data from the TCP connection. Eventually, it will receive data from the router. Then the Router gets the data from the proxy's write pipe, and finally the proxy can write the data from the TCP buffer. Using this technique no additional buffering is necessary. Furthermore, when the TCP buffer in the proxy fills up because the router is busy, TCP automatically slows down adjacent nodes, which is desirable. When the router starts picking up packets again from the proxy, TCP will trigger a slow start.

Resilience Every node has a watchdog process that restarts the STONe Router in case of a failure or timeout. There are several causes for failures or malicious attacks:

(i) *STONe Proxy fails:* The STONe Router restarts the Proxy. The Router has all required state information to restart the Proxy to reconnect to STONe.

(ii) *STONe Router fails:* The watchdog process restarts the STONe Router. In this case all state information including any buffered data, will be lost. The Router restarts the Proxy.

(iii) *T-START packet times out:* When the T-START packet does not arrive in time the Router terminates and the watchdog process restarts the router to rejoin STONe.

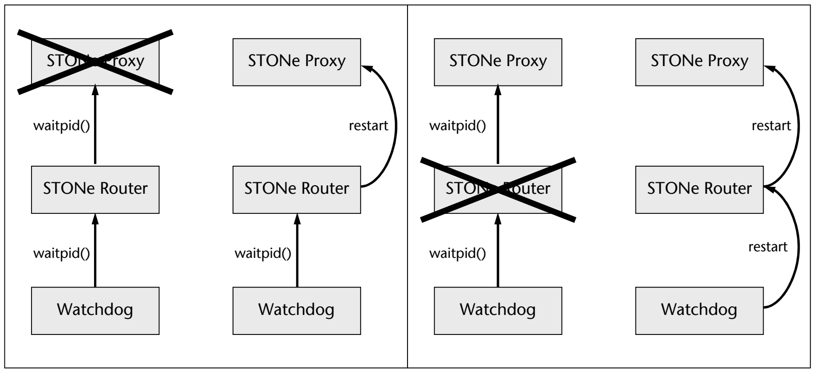

Figure 5.5: Resilience in STONe: When the Proxy crashes (left) the Router restarts it. When the Router crashes (right) the Watchdog process restarts everything.

When a node joins the network it waits until it receives the signal that it can start to send. Then it starts sending packets. When the joining node does not receive the start signal within a given time interval it time out and restarts itself.

Cryptography Most cryptographic operations take place within the TCB, but some require application-level support. STONe uses the OpenSSL cryptography library for these operations. After the Diffie-Hellman key exchange a node i has $a := DH^t_{STONe_i}$ and $b := DH^t_{STONe_j}$ and needs to compute the secret shared key for the stream cipher $K^j_{STONe_i}$. In addition, STONe must encrypt and decrypt data on the link between i and j using $K^j_{STONe_i}$. It uses RC4 and makes sure to initialize the cipher correctly to avoid possible security leaks from the key initialization procedure [81, 126].

Application-level support The Router provides sockets and pipes to the application for interprocess communication. When an application links the socket library it maps a STONe socket directly onto a Unix socket to communicate with the Router. Internally, a new STONe Socket gets a new context data structure that contains the necessary protocol status information for TSS and TDS. This data structure is available in the router and in the socket library. The socket library contains a similar data structure that is restricted to the information that is available to any untrusted processes outside the TCB.

In addition to the C library, STONe provides a Python module for STONe Sockets to make already existing classes in Python easily accessible for STONe Sockets. It is straightforward to change most socket applications into STONe Socket applications, since only the data structure of the network address changes. In addition, domain name server (DNS) requests need to be changed into requests for STONe's Trusted Name Service (TNS). Select statements on trusted sockets require STONe's tselect command.

```
int fd2cap(int fd, cap_t *cap);

void get_local_cap(fd_t rtfd, ton_service_t svc, cap_t cap);

int init_app(char *stone_appsock);

int tfd_zero(tfd_set_t *fds);
int tfd_set(int fd, tfd_set_t *fds, int type);
int tfd_clr(int fd, tfd_set_t *fds);
int tfd_isset(int fd, tfd_set_t *fds);
int tselect(int n, tfd_set_t *readfds, tfd_set_t *writefds,
                   tfd_set_t *exceptfds, struct timeval *timeout);
int tsocket(int protocol);
int tbind(int s, fd_t rtfd, ton_service_t svc);
int tconnect(int s, fd_t rtfd, cap_t dest);
int taccept(int s, cap_t *src);
int tlisten(int s, int backlog);
int tsendto(int s, void *sbuf, int len, cap_t dest, int flags);
int trecvfrom(int s, void *rbuf, int len, int flags, cap_t *from);
int tsend(int s, void *buf, int len, int flags);
int trecv(int s, void *buf, int len, int flags);
int tread(int s, void *buf, int len);
```

Figure 5.6: Trusted Socket API

Figure 5.6 shows the STONe Socket API. Most functions are equivalent to normal Internet socket calls. In addition, STONe requires its own event-handling functions that are semantically equivalent to Internet sockets, because internally a STONe Socket has two file descriptors from two different pipes.

`init_app()` initializes the application with the STONe router. `init_app()` uses as input parameter a Unix socket identifier, which describes the communication channel with the STONe router.

5.4 Applications

We have implemented three prototype applications that demonstrate the usefulness of STONe's API. The Trusted Name Service is one of the STONe's building blocks, but it also builds on STONe's API. The other two applications the are implemented so far are the Trusted Instant Messenger and the Trusted File System.

5.4.1 Trusted Name Service

The Trusted Name Service (TNS) consists of a client library and the TNS server. Figure 5.8 shows the TNS Client API. A TNS Client can either register a name and public

```
initapp(socket, n) -- initialize the application with a Unix socket address
                      and a service offset
taccept() -- accept a connection, returning new socket and client address
tbind(addr) -- bind the socket to a local address
tclose() -- close the socket
tconnect(addr) -- connect the socket to a remote address
tlisten(n) -- start listening for incoming connections
trecv(buflen, flags) -- receive data
trecvfrom(buflen, flags) -- receive data and sender's address
tsend(data, flags) -- send data, may not send all of it
tsendto(data, flags, addr) -- send data to a given address
tns_register(rt,name,key,dest,async) -- register name and key from name server
tns_query(rt,name,key,async) -- query name and key from name server
```

Figure 5.7: TSocket Python Help page

```
void tns_register(fd_t rtfd, char *name, ton_key_t *pk, int exp, cap_t csvc,
                  int async);
void tns_unregister(fd_t rtfd, cap_t csvc, int async);
void tns_query(fd_t rtfd, char *name, ton_key_t *pk, cap_t *dest, int *dlen,
               int async);
```

Figure 5.8: Trusted Name Service Client API

key with a capability, unregister a capability, or query the capability of a name and public key. TNS Client calls can be synchronous or asynchronous. Because communication is acknowledged, the client spawns a thread in the asynchronous case when the applications is not blocked. When the client registers a new TNS entry this thread also solves the challenge from the server.

Figure 5.9 shows the header of a TNS packet used for communication between client and server. `reqid` tells the server to either register, unregister or query an entry. `num_entries` contains the number of TNS entries within the packet.

5.4.2 Anonymous Instant Messenger

We demonstrate the usability of STONe's connectionless TDS service by an Instant Messaging application. The Anonymous Instant Messenger uses TNS for registering pseudonyms. Whenever a person logs in to the Instant Messenger she registers her name and public key at the name server, and when she logs off she unregisters it. For example, Alice registers the name "Chat Alice". When Bob wants to connect to Alice he requests the capability "Chat Alice" from TNS along with her public key. Alternatively, it is also possible to use Anonymous Instant Messenger as a public chat room. In this case all users register a common name such as "Chat" with the Instant

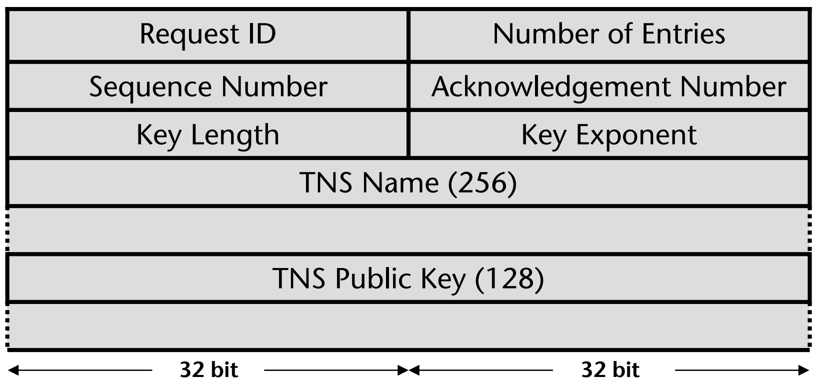

Figure 5.9: TNS packet header

Messenger and use their public keys as pseudonyms.

5.4.3 Anonymous File System

The Anonymous File System shows how the TSS service works in STONe. Anonymous File System is a simple distributed peer-to-peer filesystem, in which every client stores and retrieves files. Every node that participates runs an Anonymous File System server and client component. Whenever a client publishes a file it stores the file locally and registers the capability with TNS. Any client that wants to retrieve the file queries TNS and gets the appropriate capability. The client then connects to the peer that has stored the file. Filenames are flat in this implementation. The public key is comparable to a mini-certificate that tells the client who published the file.

Chapter 6

STONe Evaluation

In this chapter we evaluate the current STONe prototype implementation – first theoretically and then on PlanetLab [38] – to verify our claims about performance and security. The evaluation on PlanetLab is crucial because STONe's random routing depends on actual network properties. There is still leeway for performance improvements. The performance evaluation in this section should only be considered as a proof-of-concept.

6.1 Security

6.1.1 Common Attacks

Compromising STONe Nodes STONe's TCB-based architecture provides robust protection against Byzantine failures. It is therefore hard for an adversary to compromise and control a STONe node by software-only attacks. Subverting a TCB requires the adversary to get local access to the TCB hardware. In practice, this means that only a small subset of nodes can be compromised. In addition, there are architectures for tamper-evidence (e.g [182]) that quickly detect such compromises. Once detected, the compromised TCB appears on a blacklist and is automatically disconnected from STONe, because the group signature scheme in Remote Attestation will fail. The root of trust (e.g. the hardware manufacturer) has to ensure that the blacklist is constantly being updated. In addition, standard techniques such as remote device fingerprinting [114] prevent an adversary from running a virtual TCB in software with secrets she extracted from a hardware TCB.

Denial-of-Service For a Denial of Service attack, the adversary may compromise machines all over the Internet, recruit them as bots, and launch DDoS attacks against single STONe nodes or a group of STONe nodes. In common DDoS attacks on STONe, the adversary targets different kinds of resources. For STONe, network bandwidth and router CPU are the most attractive properties to attack. Bandwidth attacks are usually expensive and in STONe become even more expensive because of STONe's resilience to single failures: Whenever congestion occurs in a destination of the routing table the node simply turns to the alternative column in the row of the routing table. This eliminates one node from STONe, but because of STONe's load balancing, to completely disable STONe, all nodes need to be attacked by bandwidth flood attacks, and this is expensive. But not only brute-force bandwidth attacks are hard to accomplish in STONe. When an adversary tries to attack the Router's computational resources directly from outside STONe by sending malformed packets, the Proxy will silently drop those packets. CPU-based denial of service attacks from within the STONe network are also difficult to carry out. STONe's socket interface automatically slows down data throughput to the STONe router, since the STONe router is a trusted process. Internal STONe messages such as routing updates are encrypted and signed, and therefore, forging of these messages is not possible. The only way to isolate a node from STONe is to carry out a bandwidth flood attack.

IP Routing Attacks If an attacker manipulates IP routing information, STONe will fail, but this is the nature of an overlay network. The assumption we make in the beginning is that the network stack is well-protected against these attacks.

Replay Attacks An adversary could try to replay a handshake sequence to pretend that the adversary is another identity. However, this attack will fail because it would not be able to compute the shared key between the two nodes.

Protection against these attacks considerably improves STONe's resilience and makes it hard for an adversary to disrupt communication. This is important to prevent sabotage on the anonymity service.

6.1.2 Traffic Analysis Attacks

In Chapter 3 we explained STONe's anonymity model and measures, the attack goals in anonymity, and the adversary's properties. Traffic Analysis protection has several different objectives: Sender Anonymity, Receiver Anonymity, Unlinkability, Locality of Nodes, and Traffic Characteristics. Despite strong TCB protection, adversaries against Traffic Analysis might still be able to control the untrusted part of an arbitrary number of nodes, including the sender and the receiver.

There are several known attacks that an adversary who pursues Traffic Analysis tries to carry out. On compromised nodes an adversary might measure and record time and location of messages and correlate this information arbitrarily. She may also use one of the above common attacks to support traffic analysis, e.g. by isolating nodes from the network through DDoS.

6.1.3 Traffic Analysis Protection in STONe

STONe uses a trusted overlay network to protect traffic analysis on the underlying Internet. In particular, it should be hard for an adversary to track assignments between IP addresses and physical nodes in the network and to determines characteristic properties of the communication channel, as mentioned in Chapter 3. Inherently, STONe already protects against some traffic analysis attacks due to the design of the STONe overlay. For example, in Trusted Overlay Networks an adversary is not able to identify IP source addresses of arriving messages. However, in STONe she is able to see the IP addresses of the immediate neighbors in the hypercube.

In addition, scalability is very important to provide better anonymity. Compared to fully connected networks such as Tor or Crowds, STONe's hypercube topology is scalable and thus better suited for maintaining large anonymity sets and therefore better anonymity. Due to the scalable and resilient hypercube structure the network scales up to a larger number of nodes, similar to what has been suggested in mix networks [67] or the Crowds-based AP3 system [127]. Furthermore, only nodes that have an incentive to provide anonymous communication stay in the network. Others leave. These system properties improve anonymity.

When a sender establishes a new path to start communication, most anonymity protocols use random walks over graphs. The main advantage for doing this is that random walks provide mixing properties without using specific mixes [146]. Mix networks shuffle messages locally, whereas random walks depend on different path lengths to shuffle messages for anonymity. In contrast to STONe, mixes synchronize traffic globally within the network. We decouple the two tasks in STONe's design – *random routing and synchronization* – and show how to optimize them separately.

Random Routing

Random walks used in common anonymity protocols have several problems. If two nodes A and B are in close proximity, with a high probability a random walk only has a short path and generates localized traffic patterns that a traffic analysis adversary is able to exploit.

In a random walk over an ideal hypercube the lower bound on messages per link can be up to $\frac{N}{\log N}$ depending on the path permutation the routing algorithm implements [49]. In addition, the random walk over a hypercube takes $\log N \log\log N$ steps until it reaches a truly random distribution. Only after that can the message be forwarded to the final destination.

Theorem 1. *A random walk over a hypercube of N nodes approaches a random distribution after about $\log N\,(\log\log N)$ steps.*

Proof. For the proof we use coupling techniques. Let's start a random walk at address $00...0$ in the hypercube and go to some random node A. The length of the address' bitstring is $n = \log N$. On every step we change a random bit out of the n bits from 0 to 1 or from 1 to 0. The random walk stops when all bits are the same.

This corresponds to the Coupon Collector's problem [196]: An arbitrary set of objects contains d distinct objects, each of which can be picked with probability $\frac{1}{d}$. The problem is to determine the number of steps t it takes to pick every object at least once. The probability E_i that object i out of the d objects is missing is

$$P(E_i) = \left(1 - \frac{1}{d}\right)^t.$$

Therefore, the probability that at least one out of the d objects is missing is

$$\sum_{i=0}^{d} P(E_i) = d\left(1 - \frac{1}{d}\right)^t \approx de^{-\frac{t}{d}}.$$

When we set this probability to p_0 we get

$$p_0 < de^{-\frac{t}{d}}$$

$$t > (d + \log p_0) \log d$$

When p_0 is small it takes about $d \log d$ steps to pick every object at least once.

On every step during the random walk over the hypercube we draw a bit position out of the n bits and a bit value out of $\{0, 1\}$. A and B set the bit position accordingly. This is exactly the Coupon Collector's problem with $d = 2 + n = 2 + \log N$ objects. □

Another known scheme for anonymous communication is sorting networks such as Batcher networks [90]. However, similar to mix cascades [58, 41], they have the disadvantage that they are not resilient against compromises because they always require a fixed number of functional nodes in the network. Furthermore, sorting networks are less efficient than random walks, since they take $O(\log^2 N)$ steps to sort N elements.

Random routing, in contrast to a random walk, uses *bit fixing* in STONe's prefix-based hypercube routing algorithm. In random routing, on average, there is only one message on a given link at the same time, while the expected path length is $2 \log N$ [187].

Self-Mixing Property Because random routing picks a different intermediate node on every message transmission, the communication channel experiences random latencies. As a result, messages get randomly shuffled, making it difficult for an adversary to find packets that belong to the same communication channel. In addition, such mixing techniques are designed to prevent an adversary from associating incoming messages with corresponding replies. An adversary who knows the communication channel of some messages can only guess which communication channel other messages belong to.

We have a set $\mathcal{N}$ of N nodes in the hypercube that uses the random routing scheme described earlier. Every phase of the protocol is non-repeating, i.e. when two routes diverge they will never meet again during the same phase. This effect occurs because hypercube routing in STONe uses bit-fixing. First, we assume that all nodes in $\mathcal{N}$ send at the same rate λ. Further, every node has $h = \frac{T}{N}$ messages to send, and every destination appears on exactly h packets that are randomly distributed across all senders. This is called a 'full h-relation' [187].

Since random routing relays every message via a random node and fixes bit by bit, the probability that the path length in a network of n nodes is k follows a binomial distribution: $P_{rdist}(n,k) = \frac{1}{2\log n}\binom{2\log n}{k}$. Accordingly, the average and maximum are $\mu_{rdist}(n) = \log n$ and $M_{rdist}(n) = 2\log n$. We note that random routing cannot be modeled as a Markov Chain, because it does not have the memory-less Markov property. The Markov property states that for any process X_t and node state i_k we have

$$Pr[X_{t+1} = j | X_0 = i_0, X_1 = i_1, ..., X_t = i_t] = Pr[X_{t+1} = j | X_t = i] = P_{ij}.$$

In other words, the transition probability at the current node only depends on the current state. STONe's routing algorithm, however, depends on bit-fixing from left to right. $P[X_{t+1} = j]$ does not depend only on $P[X_t = i]$ but also on how many states the algorithm has already traversed.

Corollar 1. *Every node $X_i \in \mathcal{N}$ in a network of $|\mathcal{N}| = N$ nodes receives a sequence of messages $f_0 f_1 ... f_{2h}$. f_i are binomially distributed in $\mathcal{N}$, $f_i \sim B(\log N, \frac{1}{2})$.*

Proof. The traffic on all edges of the network is uniformly distributed for two reasons. First, we assume that all nodes in $\mathcal{N}$ send at the same rate λ. Second, every node picks a random intermediate node for every message it sends. . As a result, the traffic on all edges of the network is uniformly distributed, and every node processes messages at rate 2λ. Because bit-fixing takes place in hypercube routing, the probability that node X_i receives a message from any given node in $\mathcal{N}$ (including itself) is distributed according to a binomial distribution. More precisely, given the distance $d(X_j, Y)$ to node Y the probability that X_i receives a fragment from node $X_j \in \mathcal{N}$ at time step $t(i)$ is $P_1(X_j) = \frac{1}{N}\binom{\log N}{d(X_j,X_i)}$ where $t(i) = \frac{i}{\lambda}$. Since every node sends $h = \frac{T}{N}$ packets and packets traverse the network twice, X_i receives $h \cdot P_1(X_j) = \frac{T}{N^2}\binom{\log N}{d(X_j,X_i)}$ random fragments from every node X_j in $\mathcal{N}$ resulting in $\sum_{j=1}^{N} 2h \cdot P_1(X_j) = 2h$ fragments.

□

We also call this the self-mixing property of random routing. Similar to the original work on random routing over hypercubes we can generalize this result and define a partial h-relation in which nodes have at most h messages to send and for every node X_i there are $h_i \leq h$ messages to send. Accordingly, X_i's individual send rate is $\lambda_i = \lambda \frac{h_i}{h}$.

Lemma 1. *In the partial h-relation every node $X_i \in \mathcal{N}$ receives a sequence $f_0 f_1 .., f_{2h}$ of messages where f_i is drawn at random with probability $p_i = \frac{h_i}{hN} \cdot X_i$ and X_i is binomially distributed.*

Proof. Everything from the proof of Corollary 1 is still valid except that the probability now is $\frac{h_j}{h} P_1(X_j) = \frac{h_j}{hN} \binom{\log N}{d(X_j, X_i)}$. As a result, $\sum_{k=1}^{N} h_k \cdot \sum_{l=1}^{N} P_1(X_l) = \sum_{k=1}^{N} \sum_{l=1}^{N} \frac{h_k h_l}{hN} = 2h$ messages. □

In some real-world scenarios, especially web browsing, the relationship between senders and receivers is often only 1:n. As this is a common problem in typical anonymity networks, it is not an issue in STONe. These lemmas show that anonymity does not depend on the relationship between senders and receivers but only on the total distribution of the message destinations.

In particular, random routing protects against most intersection attacks. In intersection attacks an adversary measures traffic load on the compromised nodes. Even though an adversary may fail to identify individual connections, it is still possible to mount intersection attacks. Despite STONe shuffling messages, an adversary can increase her chances of breaking sender/receiver anonymity when senders transmit messages at different rates λ_i. The success probability $p(k, m)$ against sender/receiver anonymity in a network of k senders and m compromised nodes is:

$$p(k, m) = 1 - \left(\left(1 - \left(\frac{k}{N} \right)^m \right) \left(1 - \left(\frac{N-k}{N} \right)^m \right) \right) = O\left(\frac{k^m}{N} \right).$$

To be successful in this attack an adversary needs to compromise at least one sender/receiver and one idle node. As a result of this, cover traffic is required to ensure that senders and idle nodes are indistinguishable. If there is not enough cover traffic to hide inactivity in the network, nodes must leave the network to protect anonymity.

In addition, to break unlinkability an adversary has to compromise exactly the sender node X_i and the corresponding receiver X_j. Detecting X_i and X_j on the communication channel is only possible when the communication channel operates at a specific common rate. The success probability $p(k, m)$ for this type of intersection attack against unlinkability is

$$p(k, m) = 1 - \left(1 - \left(\frac{N-1}{N} \right)^m \right)^2 = O\left(\frac{1}{N} \right).$$

Hence, an adversary does not gain significant advantage when she tries to break unlinkability in that fashion.

In addition to intersection attacks, timing attacks pose a significant threat that must be considered as well. In timing attacks an adversary measures the inter-arrival times between two messages in sequence. For example, when a sender transmits messages f_1, f_2, and f_3 at times $t_{enter}(f_3) > t_{enter}(f_2) > t_{enter}(f_1)$, respectively, on the Internet, the time differences between the messages remain the same. In STONe, however, the time differences change along the communication path because of STONe's self-mixing property. If all senders are sending traffic at the same rate, an adversary is only able to see two consecutive fragments from exactly one node: The sender that is closest to the node.

In hypercube routing a node with distance s from the sender forwards 2^{-s} of this sender's messages. In a fully connected network only every N-th message would come from the same node, but in a hypercube there is small bias towards the closest sender. If we assume that all senders send at the same rate, one node i will receive every $(2^{-l \cdot s_j} + \frac{1}{N})$-th messages from sender j. In hypercube routing the nodes closest to the sender propagate $\frac{1}{2}$ of the sender's messages, the second closest $\frac{1}{4}$ etc.

The attack is successful against unlinkability when an adversary manages to compromise the sender, receiver and one of their closest neighbors for each. The problem, however, is that not all neighbors forward a different number of messages from the sender. For example, the probability that the first neighbor forwards two sequential messages is $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. For the second neighbor it is $\frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$. In total, the probability that two messages are arriving in sequence when the sender and its neighbor are compromised is

$$q(N) = \sum_{i=1}^{\log N} i \cdot 4^{-i} = O\left(\frac{1}{N}\right).$$

On the other hand, the probability that at least one sender and one of its closest neighbors are compromised is

$$p(N,k,m) = \left(1 - \left(\frac{N-k}{N}\right)^m\right)\left(1 - \left(\frac{N - \log N}{N}\right)^{m-1}\right) =$$

$$O\left(1 - \frac{(N - \max(k, \log N))^m}{N^m}\right).$$

Here, k is again the number of senders and m the number of compromised nodes. Given that the send rate λ must not be uniform for a successful attack, the success probability depends on the send rate as well, and there are $\frac{\lambda}{\lambda_j} \log N$ neighbors to consider instead of $\log N$. Timing analysis can be a threat to STONe, because the overall success probability $q(N) \cdot p(N,k,m)$ is not necessarily small. To fully protect against timing analysis attacks all senders have to be synchronized.

Attacks on the Topology When an adversary knows STONe's hypercube topology, timing analysis attacks may become significantly easier. In particular, when an adversary knows the sender's neighbor, on average, she is able to monitor every N-th message and correlate these measurements across the network. Therefore it is crucial that STONe hides network topology information.

By design, STONe hides network topology, since an adversary can only access the routing tables from within the TCB. Further, STONe derives node addresses from the internal Trusted Computing keys, thus randomizing them. A STONe node address is unrelated to its location in the network.

However, there are still ways to recover the topology due to physical network properties. The main adversary against the hidden topology is tomography. In tomography a network monitoring tool typically probes network end-to-end delays to infer individual communication path characteristics (e.g. [60]). If an adversary detects these individual characteristics she may be able to partially reconstruct STONe's topology.

Typical tomography works the following way: First, an adversary uses all compromised nodes to determine the latencies between them. In the second step she connects to a random node in the network and measures the end-to-end delays. The lower bound of the end-to-end delay is a measure for the distance between the node and the current node. Usually, there exist algorithms that are able to reconstruct latencies of all possible $O(N^2)$ paths using only $O(N \log N)$ path measurements [60].

In the next theorem we give an upper bound for the path length. For simplification we assume that all links have uniform delays, and end-to-end delays only depend on the path length. For simplification we assume that the hypercube routing algorithm uses $k = 1$ and $l = 1$ as parameters.

Theorem 2. *(Valiant [187]) In random routing the average number of hops along an arbitrary path is $\mu = \log N$. When all nodes send h packets the probability that a message does not get delayed by more than $\Delta + \mu$ steps is:*

$$P(X \geq \Delta) < hN \left(\frac{eh \log N}{2\Delta} \right)^{\Delta}$$

On average the path length in random routing is $2 \log N$. If we set $\Delta = k \cdot \log N$ then $P(X \geq k \log N) < N^{-k}$ when $k \geq eh$. For a simple permutation $(h = 1)$, this bound holds for $k \geq e$.

Lemma 2. *The expected arrival-time variation between two fragments in a sequence is $\sqrt{\frac{\log N}{2}}$.*

Proof. The distance between sender and receiver is binomially distributed with distribution $B(2 \log N, \frac{1}{2})$. The expected variation in distance between two independent trials is the standard deviation: $\sqrt{2\log N\frac{1}{4}} = \sqrt{\frac{\log N}{2}}$. □

STONe minimizes congestion even in the worst case, when the network routes messages synchronously. However, STONe's routing buffers have to tolerate maximum delay. End-to-end latency can grow up to $14\log N$ hops depending on the permitted loss rate [47, 187].

So far we have only investigated tomography in a network with homogeneous links. For STONe we also have to consider tomography in hetereogenous networks that have individual link delays such as the Internet.

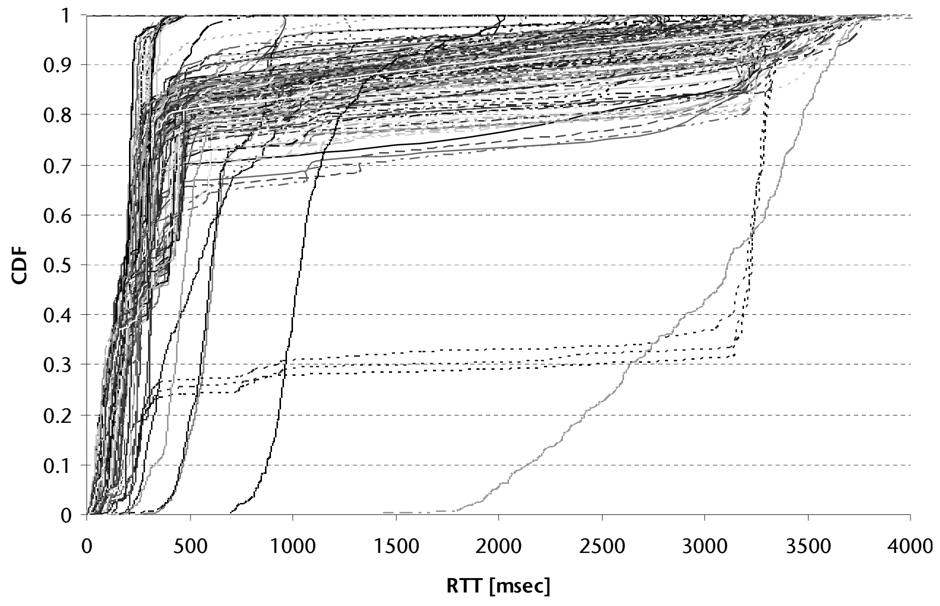

Figure 6.1: Cumulative Distribution Function of an all-pairs-pings on PlanetLab: A single curve represents the CDF of the RTT between a single node on PlanetLab and the rest of the network.

We first conduct an all-pair ping on PlanetLab to measure the round-trip times between all pairs of nodes. PlanetLab is the current testbed for distributed Internet applications [17]. It emulates the real-world Internet, because it allows applications to run on geographically distributed machines that have different computation and communication capabilities [38].

Figure 6.1 shows the cumulative distribution function (CDF) over an all-pair ping measurement. Every single graph represents the RTT between a fixed node and another arbitrary node on PlanetLab. On average only a few nodes have large RTTs, whereas 80% of the nodes have RTTs below $500msec$. Some nodes in the graph are weakly connected.

Because RTTs are heavily distributed, an adversary can use additional knowledge about RTTs to reconstruct the topology. After discovering the overall path length l

and delay D, the adversary's goal is to detect the specific individual link delays d_i along the path from all $N \log N$ individual link delays in STONe. Once she knows the individual link delays she can use the results from tomography to determine the links within STONe that are included in the path.

The number of possible different combinations of summations of individual delays d_i resulting in D can be computed by a polynomial of degree $(l-1)$ if we assume that all individual link delays in STONe are distinct. For example, when $l = 1$ there exists exactly one possibility, $D = d_0$. When $l = 2$ there are about $\lfloor \frac{(D-1)}{2} \rfloor$ possible summations. When $l = 3$ we have to sum up over all possibilities of length $l = 2$. We can express this recursion in closed form q_l in which y_i describes the number of summations for path length i that result in end-to-end delay x:

$$\frac{q_l(x)}{(\log N)!} = \frac{(1 - y_{l-1}x^{l-1})(1 - y_{l-2}x^{l-2})(1 - y_{l-3}x^{l-3}) \cdots (1 - y_1 x^1)(1 - y_0)}{(\log N)!}$$

Now, the probability that an adversary gets a correct combination of individual delays whose sum results in D is:

$$P_{delay}(D, N, l) = \frac{q_l(D)}{l! \cdot 2^{N \log N}}.$$

D does not depend on N and in STONe $l < 2 \log N$ for most practical cases. When N becomes large the success probability is very small and therefore a brute-force attack by random guessing is hard.

This specific problem can also be specified as a subset sum or knapsack problem. Knapsack problems can be solved under certain circumstances [136], in particular, when knapsacks have low density. If the d_i are chosen at random with $d_i \approx 2^{\beta N}$ and $1 \leq i \leq N$ where $\beta > 1.54725$ then the knapsack is easily solvable. However, this distribution does not apply to link delays on the Internet. Furthermore, link delays have large variance, and therefore, they cannot be formulated as a low density knapsack problem.

Limitations of Random Routing

A significant trade-off in random routing is that only half of the bandwidth in the network is available. Further, on average the round-trip time becomes twice as large. Furthermore, random routing is still prone to intersection attacks when synchronization does not take place, and there are idle nodes in the network.

Fragment Sizes

Even when an adversary only has access to a few nodes she can observe unique fragment sizes, and by doing traffic confirmation attacks she can assign traffic characteristics to particular communication channels [183].

A common solution to the problem is to use uniform fragment sizes. However, uniform fragment sizes have a large overhead.

An alternative way to deal with the problem is message splitting [164]. STONe could split a message that would otherwise fit into one fragment into smaller pieces and route them through the network along different routes using random routing. When we split every message into N fragments of random size every node observes a random fragment every time. However, this also gives an adversary an estimate for the upper bound of the message size.

Ideally, the fragment size is a global synchronization parameter that depends on the session-layer requirements of the different communication channels. Every STONe node uses a uniform fragment size in the beginning. When the real fragment size is smaller it decreases the size by some maximum Δ, otherwise it increases the size by some maximum Δ. In random routing every node sees the different fragment sizes and adjusts the uniform fragment size to the average measured fragment size. We leave this as subject for future research. In the current implementation STONe uses constant fragment sizes.

Session-based Attacks When a session expires, counting attacks become a threat because the inactive node is part of the anonymity set but does not show any activity. There are two alternatives to fix this problem: Either the node drops out of the network, or it has to start some activity.

When a node leaves the network it reduces the size of the anonymity set. On the other hand, when a node has to start activity that is not intended by the user it costs extra resources. STONe has to make a trade-off between the *churn rate* $\lambda_{churn} = \lambda_{enter} - \lambda_{leave}$, i.e. the rate at which nodes are entering and leaving the network, and the *average send rate* $\bar{\lambda}$, at which nodes are sending traffic within the network.

In fact, when a single node enters and exits the network up to $\log^2 N$ routing update messages have to be sent. In contrast, sending extra traffic requires up to $\bar{\lambda}$ of extra data, where $\bar{\lambda}$ is the average data rate in STONe. Optimizing this problem is another important future aspect of research.

6.1.4 Application-based Anonymity in STONe

Now we look into STONe's anonymity on the application-layer. Here, STONe provides the Trusted Name Service as well as STONe sockets as the main features to prevent an adversary from spying on network addresses and possible communication patterns.

Application-based Denial-of-Service STONe has to protect against a misbehaving application that tries to flood the network. However, STONe has the same semantics as TCP/UDP sockets. When an application sends too much data, filling the buffers quickly, STONe will block. When a node opens too many connections STONe will eventually run out of memory. STONe has some advantages over TCP/UDP sockets because it can shut down a node that tries to launch a DDoS attack, similar to SOS, I3 or Mayday [110, 29, 179].

Denial of Service on TNS An adversary could try to register a bulk of TNS_{Name} entries and brings the server down. There are two issues: First, TNS gets flooded with new entries, and second, TNS can use up all of its memory, when it has to keep track if half-open entries. TNS uses SYN cookies to avoid this problem [20]. Further, TNS does not spend any CPU time on encryption until it has got a response from the registering STONe client node. To solve the first problem, however, STONe requires some form of admission control. One solution is to limit the number of entries per node, but that would not work in TFS because file servers register significantly more entries than other nodes. Another solution is to have timeouts on the entries and use computational client puzzles, so that TFS can foresee how many entries to expect from any given node.

Side-Channel Attacks An adversary cannot directly attack the STONe system, but she is able to monitor system activity. There are possibilities that system load or kernel activity reveal some patterns that can compromise the anonymity properties of the application. As suggested in Chapter 2 extra dummy load on the nodes makes it harder to mount these attacks.

Impersonation Attack Because TNS does not certify names but only public keys, STONe applications need to have secure offline channels or key escrow for obtaining the correct identities in STONe. Otherwise it is always possible to impersonate someone else on TNS. But this is not much different from standard certification authorities like Verisign [22] that do offline checks once.

Censorship Attack Since STONe does not have a central membership list it is hard for an adversary to disconnect or launch a DDoS attack on single nodes. On the application-level TNS is the most vulnerable point because nothing works without the name lookup service, similar to DNS in the Internet. However, since TNS is a trusted process it is hard for an adversary to shut it down, but it is prone to DDoS attacks. TNS has to be replicated to improve resilience and also performance. In particular, the distributed applications like TFS require a lot of name lookups.

Passive Logging Attack Since STONe's addresses are opaque capabilities and traffic analysis is hard, even a global adversary with access to sender, receiver, and arbitrary nodes does not learn much about communication channels unless he is able to compromise the TCB, which can happen only at a very rare occasion. Therefore, Passive Logging Attacks in STONe are not so powerful.

Phishing and Pharming Attack The Trusted Name Service in STONe is strongly protected against attacks from the TCB. And this TCB protection makes any type of Pharming attack hard. Additionally, the security model in STONe is different from the Internet. DNS maps real domain names to IP addresses, whereas STONe maps pseudonyms to opaque STONe capabilities. In contrast to DNS, which has a hierarchical name space, pseudonyms are unrestricted. Eve can only register an arbitrary pseudonym and STONe guarantees that she has the corresponding public key. Everything else depends on the "web of trust" in STONe [86].

However, when Eve wants to start a Phishing attack to fool Bob that she is Alice she needs to set up a web server that contains the fake web site. This web site needs to carry fake credentials from Alice. But because the trusted name server verifies Alice's public key, Eve is only able to impose Alice's identity when she knows Alice's private key. Therefore, Phishing attacks are almost impossible.

6.1.5 Anonymity Goals

Summarized, STONe achieves the following individual anonymity goals:

Membership Anonymity In common anonymity systems that consist of untrusted proxies such as Tor [75] or mix networks [58] the clients themselves are only known to one single proxy. However, often the set of proxies $\mathcal{M}$ is public, and an adversary can directly focus on blocking or attacking these proxies. STONe does not have to distinguish between clients and proxies for anonymity reasons, because the membership list $\mathcal{M}$ itself is hidden inside the TCB. Monitoring packets entering and exiting a single STONe node just provides an adversary with information about $S(N) = O(\log N)$ random nodes out of N.

Traffic Anonymity On the Internet network packet content, packet size and inter-packet delays reveal traffic characteristics per se. Existing anonymity networks for low-latency communication such as Tor or Crowds [75, 148] do not implement any techniques that protect against all of these problems. STONe, however, inherently disguises inter-packet times. Random routing causes packets to take different random routes through the network, and therefore end-to-end delay is random as well. STONe also makes network packet sizes uniform and re-encrypts packet content using different keys on each hop. An attack on traffic anonymity in STONe needs to be more sophisticated and typically involves sender and receiver compromises which are hard to achieve.

Sender/Receiver Anonymity Sender and receiver anonymity in typical anonymity networks focuses on detection of exit nodes, because exit nodes have information about the clients participating in the communication. One of the most efficient ways to attack sender/receiver anonymity is the predecessor attack [198]: When path reformations take place, sender and receiver never change but intermediate nodes along the path do. If an adversary samples the source addresses of arriving packets on random nodes she observes the sender address more often than addresses of intermediate nodes. It is therefore possible to identify the sender and receiver. Frequent path reformations therefore significantly increase the risk of a breach in sender/receiver anonymity. However, if the adversary is unable to identify the connection it is not possible to successfully link together detected senders and receiver on the network.

STONe prevents the adversary from identifying the connection, because the message include the packet header is encrypted, the packet size is uniform and the path itself is not distinguishable from any other communication path on the network due to random routing. This clearly prevents an adversary from carrying out the predecessor attack.

The best strategy for an adversary to compromise sender/receiver anonymity is to monitor traffic properties at the sender and the receiver nodes directly. When, for example, the total measured traffic volume over time is equal at two given nodes it is likely that they are connected. Also, an adversary may make use of additional knowledge about the network topology to carry out these intersection attacks. To protect against such attacks it becomes necessary to synchronize the network at least partially and cover the sender and receiver nodes.

Figure 6.2 shows how STONe's anonymity degrades under random routing when groups of 2, 4, 8, 16, and 32 nodes are synchronized. When only 2 nodes are synchronized the network is in the same state as if no synchronization takes place. However, by adding only a little synchronization it is already possible to slow down anonymity degradation significantly. Random routing supports efficient implementation of synchronization, because every node knows the total traffic volume of the network.

As a result, STONe is resistant against most traffic analysis attacks. Most importantly, STONe prevents the Predecessor Attack, because an adversary is not able to identify network connections, thus allowing frequent path reformations. STONe is also scalable and therefore supports large anonymity sets. Its core benefit in protecting against traffic analysis lies in the fact that it uses random routing instead of random walks used in previous approaches to anonymous communication. Random walks still preserve locality and therefore do not provide optimal anonymity.

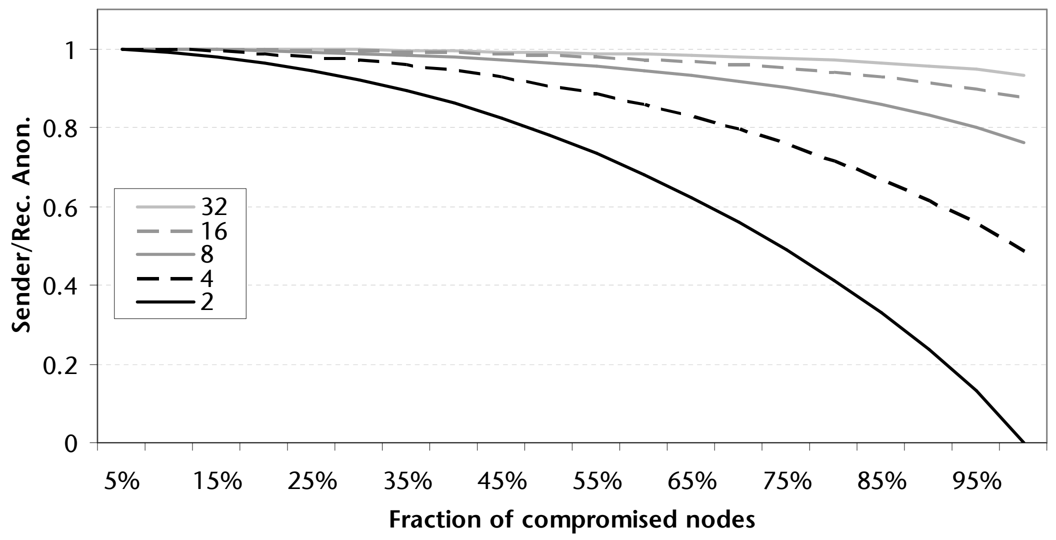

Figure 6.2: **Sender/Receiver Anonymity Degradation:** Partially synchronized nodes under Random Routing in STONe stop the sender/receiver anonymity degradation.

6.2 Performance

Our theoretical security analysis showed that STONe's features defeat many attacks. However, we need to verify that the actual implementation indeed realizes the desired properties. First of all, we investigate STONe's performance overhead.

6.2.1 Microbenchmarks

In the first series of experiments we measure microbenchmarks on two STONe nodes. STONe runs on two Linux machines that are connected via 100MBit/s Ethernet. One machine has two 3.2Ghz Pentium 4 CPUs with 2GB memory, the other two 3GHz Pentium 4 CPUs with 1GB memory. Both run Fedora Linux kernel version 2.6.9. The goal is to measure the baseline performance overhead of STONe.

Hop-by-hop Latency The objective of the first experiment is to determine hop-by-hop latency between the two nodes over Ethernet for various fragment sizes. The round-trip time (RTT) of a common ICMP ping is the bottomline case for the RTT between the two STONe nodes. In this scenario STONe carries out full encryption but no random routing or synchronization. In the experiment the message size varies from 512 to 4096 bytes. 4096 bytes is the maximum, since this is also the maximum capacity of the internal IPC pipeline between the STONe proxy and the STONe router.

Figure 6.3 shows the results of this first experiment. STONe has a constant processing overhead of about 150 μsec for every message and an additional 2-5% for encryption.

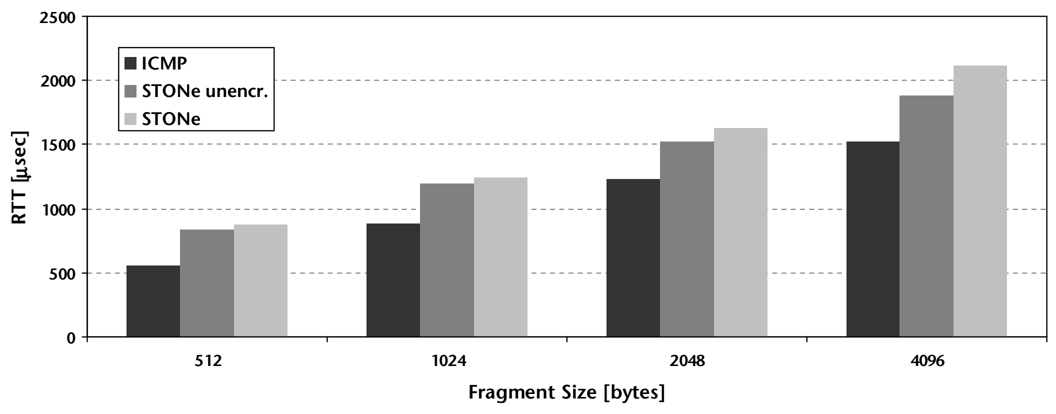

Figure 6.3: Hop-by-hop round-trip times for different message sizes: Both RTTs for ICMP and STONe increase at the same rate as the fragment size. STONe's overhead is approximately 2-5% for encryption, and a fixed overhead of 150 μsec for the additional routing layer on every node.

Proxy overhead	40 μsec
Router overhead	60 μsec
Context switch and IPC	40 μsec

Table 6.1: Processing overhead in STONe: Look-ups in the routing tables make up the largest share of message processing time. The remaining time is spent by the proxy and the OS-based scheduling and context switches.

Table 6.1 shows the time spent during the message processing. A message spends most of its time in the router when it looks up the next hop. The proxy itself has system calls for reading and writing data, and the rest of the time is spent on context-switches and message copying between the proxy and the router.

The two-process architecture in STONe is responsible for some of the overhead. A single process could reduce the overhead by 50-100 μsec. However, the router would not be protected from outside attacks, especially when the network stack is under a DDoS attack. In our current architecture the router monitors traffic patterns at the Proxy and automatically detects when it is being attacked. It can then start another Proxy that listens to a different port. For performance reasons it may also be worthwhile to use kernel threads instead of user-level processes. However, the current PlanetLab architecture has constraints that prevent the use of kernel threads.

End-to-End Throughput In the second experiment we determine STONe's end-to-end throughput between these two nodes. We measure the average throughput during a large file transfer from one node to the other via the Trusted Stream Service (TSS). In the experiment we change the maximum window size as well as the fragment for every different run.

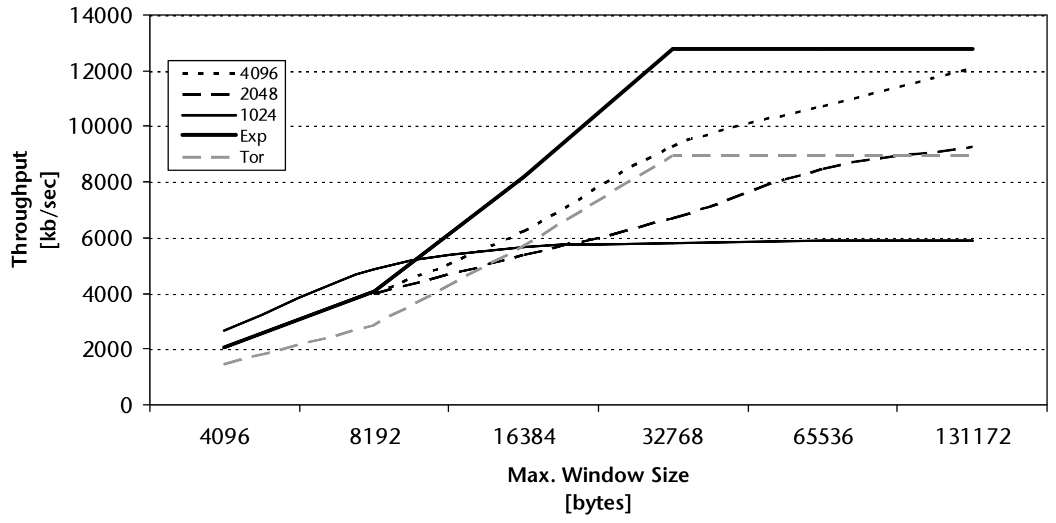

Figure 6.4: End-to-end throughput in STONe for different fragment sizes of 4096, 2048 and 1024 bytes compared to the expected throughput of the connection (Exp) and the measured Tor throughput (Tor): For large fragment sizes STONe achieves almost the expected throughput for every maximum window size. Smaller fragment sizes require more processing overhead.

Figure 6.4 shows the measured throughput as a function of the maximum windows size. The expected theoretical throughput is $\frac{max\ window\ size}{2 \cdot RTT}$ or the maximum bandwidth of the connection, whichever is smaller. The RTT between the two nodes is $100\mu sec$ and the bandwidth 100 MBit/sec. Therefore, the maximum possible throughput is about 12.5 MB/sec.

TSS stalls when the maximum window size is smaller than four fragments, because it does not saturate the underlying TCP connection, and TCP slows down. STONe achieves almost the expected throughput between two nodes, especially when the send window size becomes large.

In comparison, the graph also contains the measured throughput of Tor by using the 'torify' command, which turns a standard socket application into a Tor socket application [75]. Tor uses TCP flow- and congestion control mechanism, which is not optimized for long delays in anonymity networks. As the graph shows Tor imposes a penalty of about 30% on throughput in general, which is significantly larger than STONe's minimum penalty under TSS.

These microbenchmarks show that STONe's prototype implementation can accommodate reasonable throughput for most Internet applications. In random routing this implementation achieves an average throughput that is close to the theoretical maximum throughput. This is subject to further improvements as for smaller packet sizes overhead for copying data between user-space and kernel-space occurs. However, we had to implement this prototype in user-space and leave the kernel-level implementation for future work, because PlanetLab as our testbed does not permit kernel modules. For end-to-end application-level communication between directly connected nodes, copying packets between user and kernel space does not make any difference. But on multi-hop routes this overhead occurs on every single hop. As an additional option STONe can reduce the number of hops on the path at additional cost for the routing table space and additional handshakes upon node join operations.

6.2.2 Basic System Performance

In the second series of experiments we run benchmarks on a large-scale environment to validate our theoretical analysis about the network topology from earlier chapters.

Scalability Scalability is crucial in STONe, since a system for anonymous communication requires a large anonymity set of nodes. In these experiments we measure the average number of routing table entries in a network of 16 to 512 nodes.

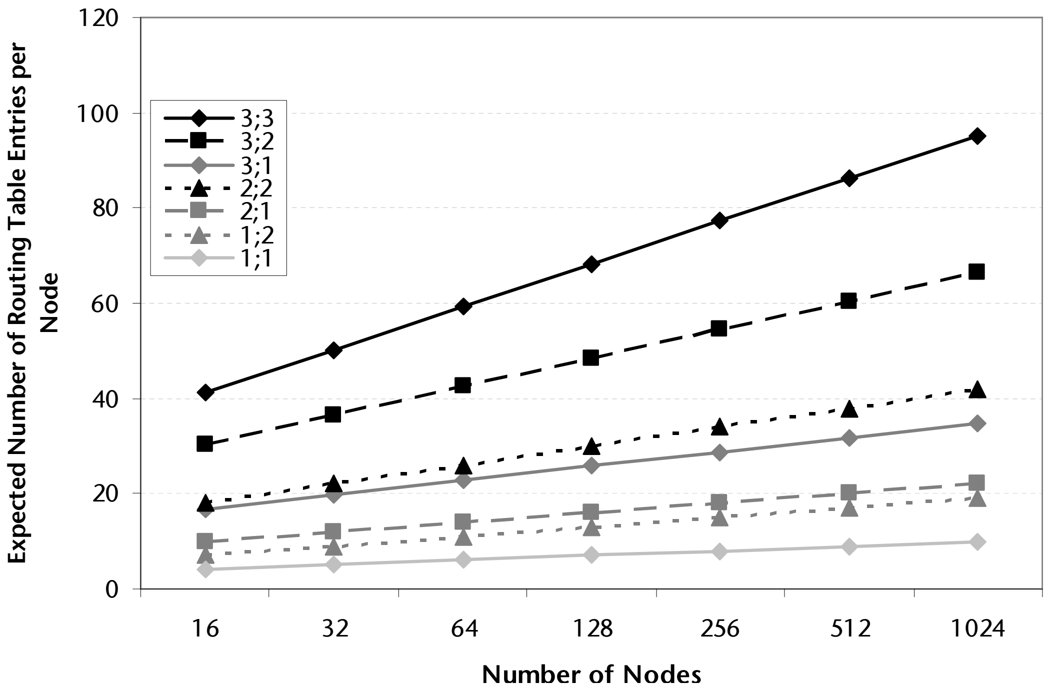

Figure 6.5: Routing Table Sizes for a variable number of STONe nodes N with $b = 2$ different values for the digit in the address and parameters $k;l$ with k virtual addresses and l alternative routing table entries per node: The measured routing table size in STONe approaches the theoretical expected value $S(N)$.

Figure 6.5 verifies that the increase in routing table size is logarithmic in the size of the STONe network. Remember, that we have k addresses per node in the overlay and l alternative entries per routing table slot with a correction factor. Therefore, the routing table size $S(N)$ is not exactly logarithmic but $S(N) = \bar{k}l(b-1)(\log_b(\bar{k}N))\left(1 - \frac{l-1}{2}\right)$.

To better illustrate the routing table sizes for different k and l we keep the number of STONe nodes N constant but only change parameters k and l. Again, we measure the routing table sizes and compare them against the expected size.

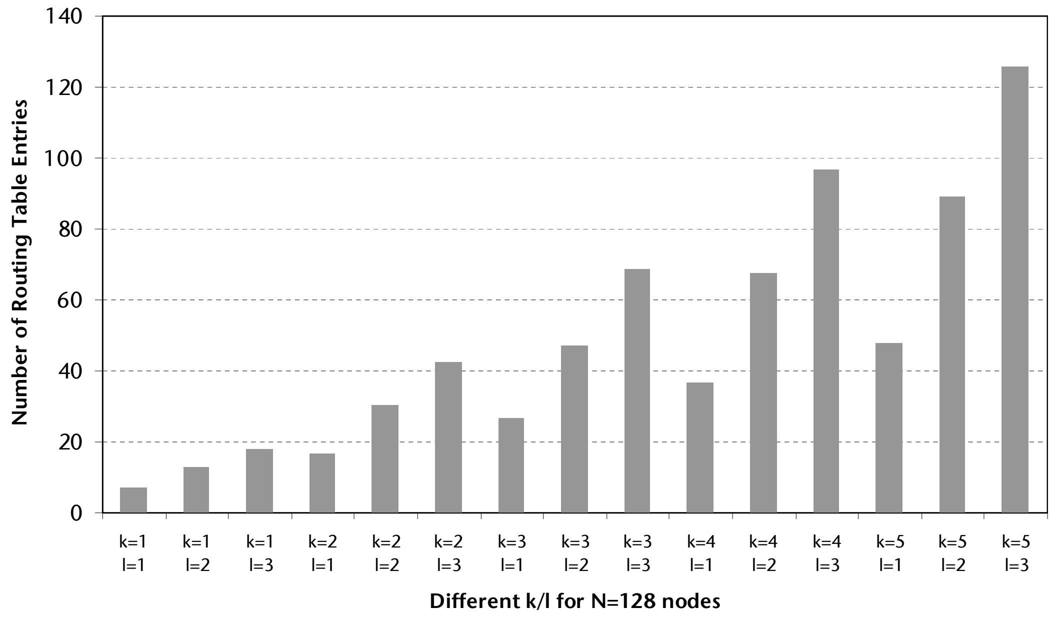

Figure 6.6: Routing Table Sizes for $N = 128$ with k virtual addresses per node and l entries per routing table slot.

Figure 6.6 contains the number of routing table entries for constant number of STONe nodes $N = 128$. When l increases the routing table size increases polynomially. When k increases the routing table size increases logarithmically. This is exactly what we expect from $S(N)$.

These results also show that STONe indeed supports high churn. Even for large networks with hundreds of nodes the number of required handshakes during an insert operation for a joining node is limited.

6.2.3 Random Routing on PlanetLab

Finally, we evaluate the outcome of random routing on PlanetLab. Remember that it is the goal in random routing to protect against timing attacks and contextual attacks, such as intersection attacks. In addition, an adversary who is observing random connections should not be able to compromise sender/receiver anonymity or unlinkability.

In this experiment we send STONe echo packets from a random node to another arbitrary STONe node. We then observe the distribution in the number of hops it takes to transmit the echo packets between the two nodes. Finally, we measure the round-trip time.

Path Length In this series of experiments we measure the number of hops between two arbitrary nodes in STONe. Random routing causes messages to take different paths through the network, and therefore the number of hops is also random. Remember, the expected path length in STONe's routing is

$$L(N) = \frac{1}{2}\log_b\left(\frac{N}{\bar{k}l\left(1-\frac{l-1}{2}\right)}\right)$$

hops.

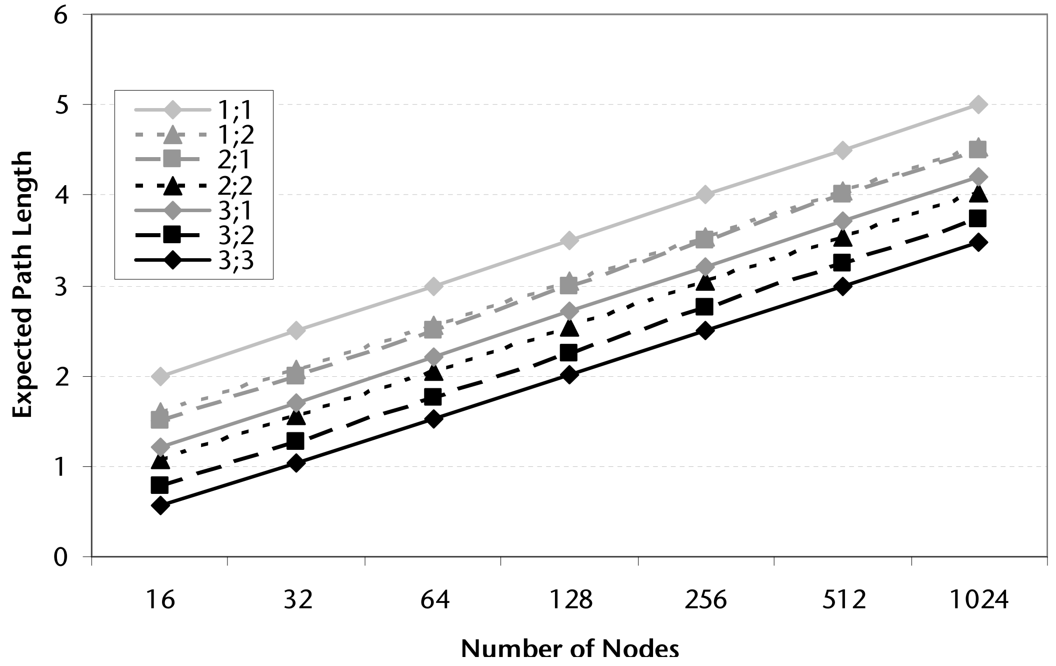

Figure 6.7: The average single path length between the same sender and receiver in random routing over time for different pairs $k;l$ with k virtual addresses and l alternative routing table entries per node.

In the first experiment we measure the average number of hops between two STONe nodes on STONe networks of different sizes. Figure 6.7 shows that we always approach the expected routing table size $S(N)$, thereby proving our theoretical results.

In the next experiment we change parameters k and l and measure the average number of hops on a STONe network of $N = 128$ nodes. Figure 6.8 shows that we almost always achieve the expected value. When k is constant and l grows, the path length decreases polynomially. When l is constant and k grows, the path length decreases logarithmically.

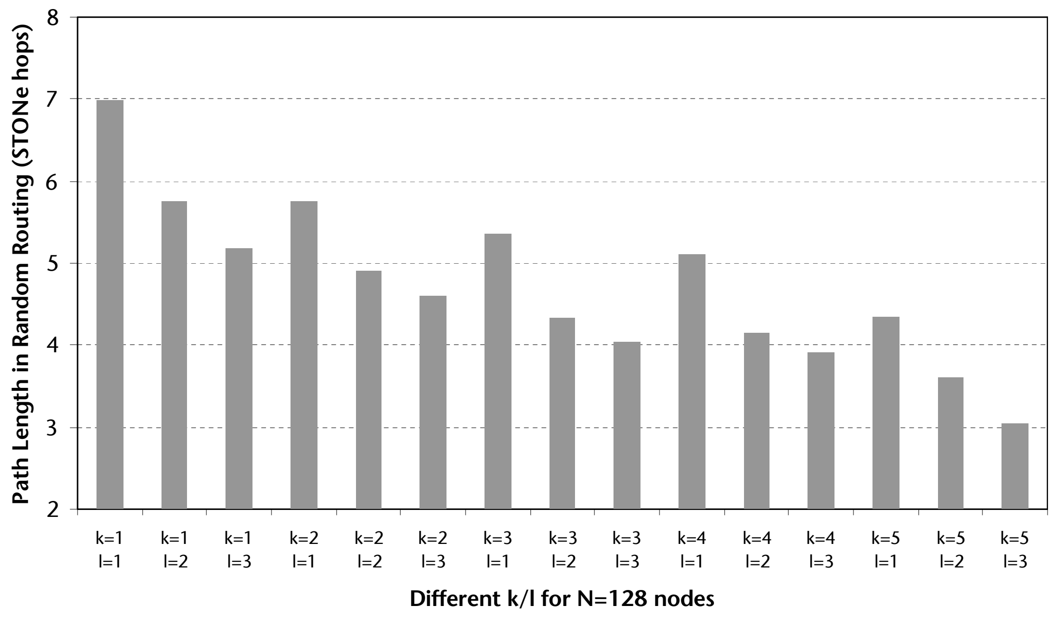

Figure 6.8: The average round-trip path length between the same sender and receiver in random routing for k virtual addresses and l alternative routing table entries per node for a STONe network of 128 nodes.

To evaluate random routing quantitatively we define a cost function $C(N)$ for the STONe network as

$$C(N) = \frac{S(N)}{L(N)^2}.$$

The rationale for the definition of the cost function is that the number of routing table entries $S(N)$ divided by the average path length $L(N)$ results in the number of nodes of extra information per prefix slot. If this is the number of hops required to reach the destination we have a balanced cost of 1, since this scenario would correspond to a brute-force routing method.

Figure 6.9 shows the graph for $C(N)$. Some combinations of parameters as for example $k = 2$, $l = 2$ – $k = 4$, $l = 1$ impose the same cost. The general conclusion is that $l > 2$ does not provide enough benefit in the case of $N = 128$. $l = 2$ seems to be the optimal solution, since some redundancy is required, and the studies in earlier chapters showed that $l = 2$ provides sufficient protection against path failures.

In the next step we not only measure the average path length but also the distribution of path lengths for different k and l. For the experiment we pick $k = 2$ and $l = 2$ for the first run and $k = 4$ and $l = 1$ for the second run, because they both have about the same cost.

Figure 6.10 shows the measured results. The first notable observation is that different parameters for k and l cause different variances. Smaller k seems to be

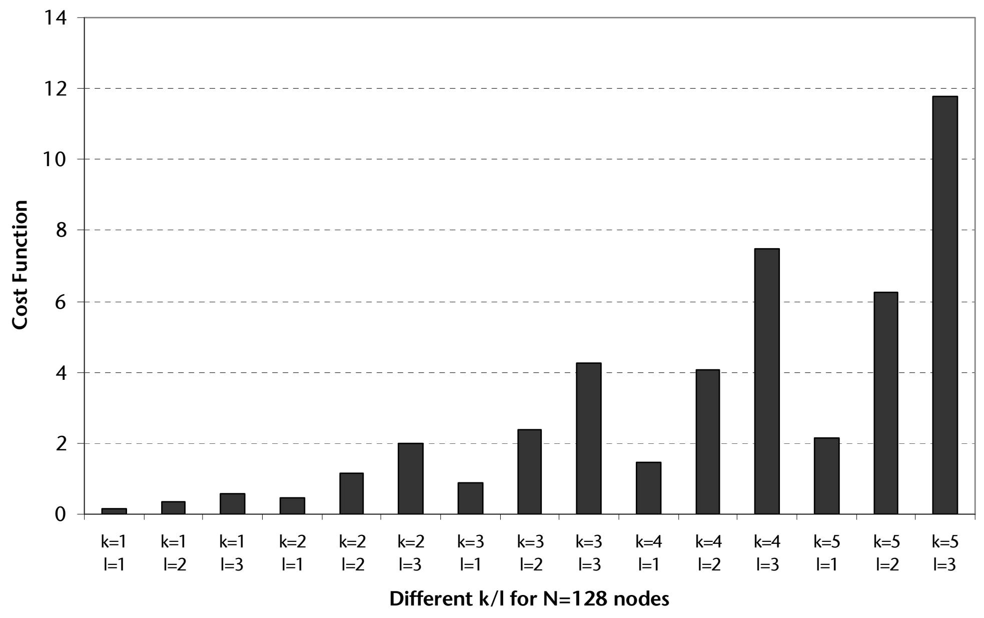

Figure 6.9: The cost of random routing for k virtual addresses and l alternative routing table entries per node in a network of N=128 nodes.

preferable to reduce the latencies. The distribution of the number of hops is modal and confirms the theory about the binomial distribution.

In the next experiment we measure round-trip times on PlanetLab for $N = 128$ and compare them against Tor. In STONe we pick $k = 2$ and $l = 2$, since this parameter pair tolerates faulty links and allows acceptable cost. In the experiment a node constantly sends messages to a fixed receiver via random routing. The receiver bounces back messages via random routing, and the sender then measures the round-trip time of the message.

Figure 6.11 shows STONe's RTTs on PlanetLab. The number of hops is binomially distributed, but the actual node-to-node delays have a heavy-tailed distribution which causes a stretch in the total round-trip times. The average round-trip time is only around $400msec$, but the maximum round-trip time is about $1000msec$.

Figure 6.12 shows the distance in the number of hops it takes messages to get from the sender to the receiver and back. As expected it is binomially distributed with a peak at 7.

Finally, figure 6.13 shows the cumulative distribution function over the hop-by-hop delays of the selected 64 PlanetLab nodes. Hop-by-hop delays are usually between $30msec$ and $120msec$, which is realistic for a continent-wide network. A transcontinental ICMP echo across the US usually takes about $70 - 100msec$.

In contrast to STONe, figure 6.14 shows the RTTs on Tor when using Tor sockets in connection with a user-level echo server over TCP. The standard deviation of the

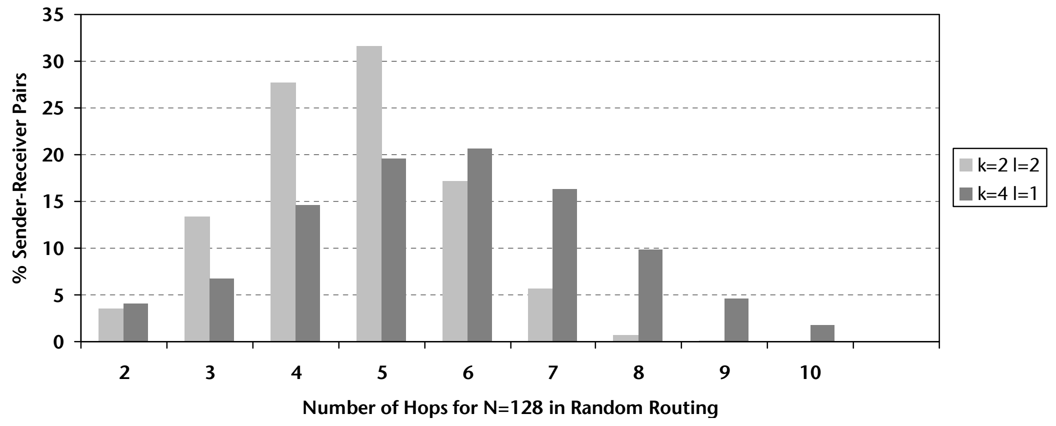

Figure 6.10: Distribution of the distance in STONe between all sender/receiver pairs for $N = 128$: The path length is binomially distributed for different k and l.

round-trip times is more concentrated around the average round-trip time, and the absolute average value is larger. Tor's problem is that it is not fault-tolerant and cannot easily switch to alternative routes when congestion occurs.

Anonymity The final experiments verify the anonymity claims of STONe. In STONe, an adversary has to eavesdrop on the sender or receiver itself to compromise sender/receiver anonymity or unlinkability. This is much stronger than what mix networks or onion routing provide.

For this experiment we count the number of messages that a single node forwards. When an adversary listens to connections leading to these nodes, this would be the data volume she is able to count. To maximize the adversary's chances we generate traffic only between selected nodes in the entire network without any background noise.

Figure 6.15 shows the result. As predicted, only the sender and receiver have forwarded more fragments than any other node, and the traffic over the remaining nodes is almost balanced because of random routing.

In the last experiment we want to test synchronization to bridge the gap between sender/receiver and the rest of the nodes. This gap is responsible for anonymity compromises. For this experiment we set up a network of 8 nodes, transfer a large file from one node to another, and measure the throughput.

Figure 6.16 shows the results. In a network of 8 nodes STONe smoothly approximates a maximum throughput. However, as shown earlier, the penalty for synchronization is large. In the beginning the actual difference between the required throughput λ_i and the average throughput $\bar{\lambda}$ is large, and therefore the increment has to be large as well. The resulting throughput $\bar{\lambda} + \Delta\lambda_{max}$ then increases logarithmically

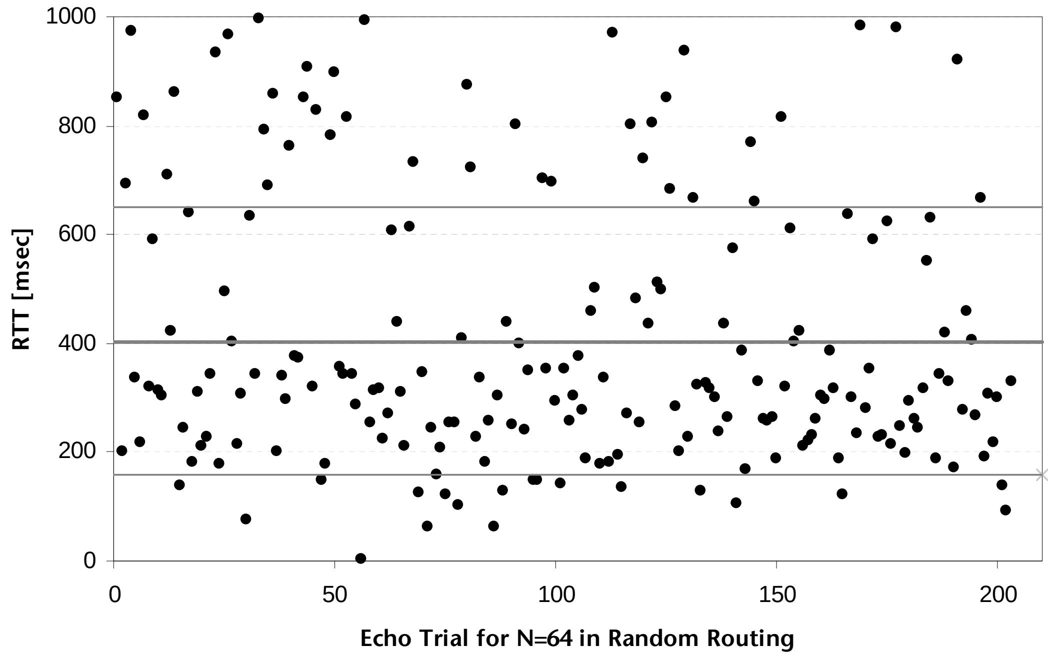

Figure 6.11: STONe's RTT between the same sender and receiver in random routing over time: It shows that random routing results in a random RTT. The middle line marks the average RTT whereas the other two lines delimit the standard deviation around this average RTT.

over time as expected. It is subject to future work to determine the optimal parameters for synchronization and to minimize its overhead in large environments.

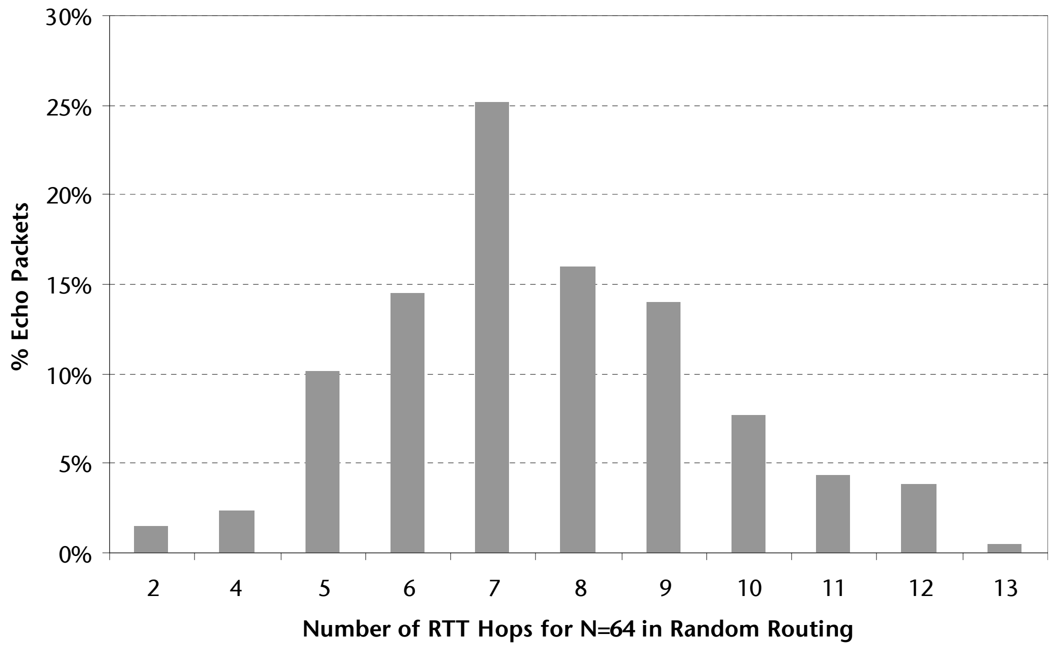

Figure 6.12: STONe's distances in the number of hops for the round-trip experiment when $N = 64$ in random routing. It shows that the distance for both ways is also binomially distributed.

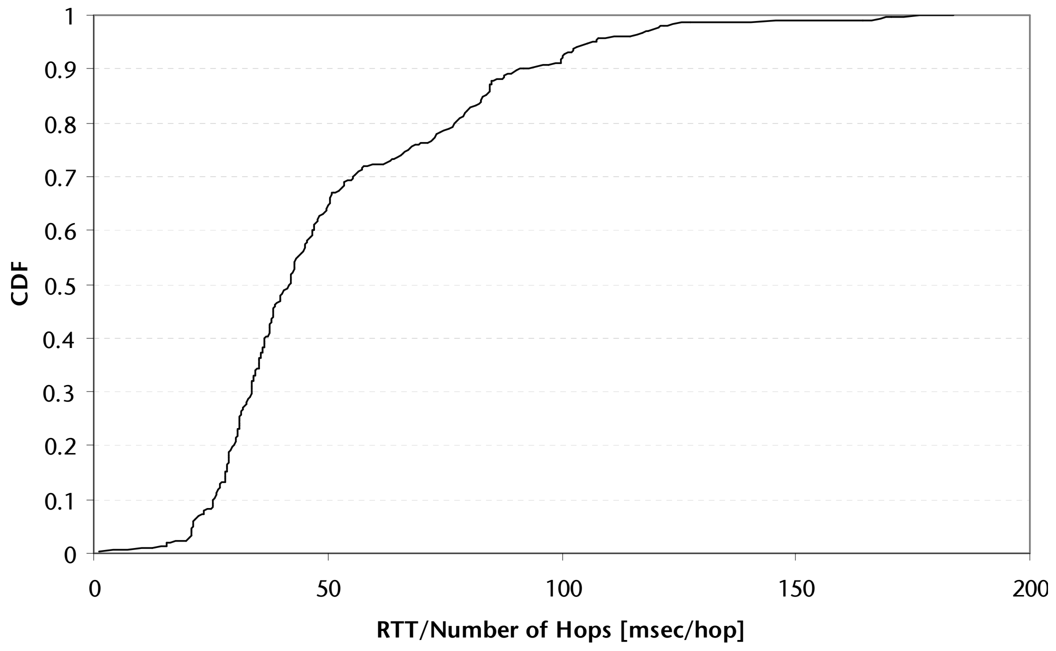

Figure 6.13: Cumulative Distribution Function of RTT/Number of Hops in random routing: It shows that the assumptions for hop-to-hop delays in the PlanetLab experiment are realistic.

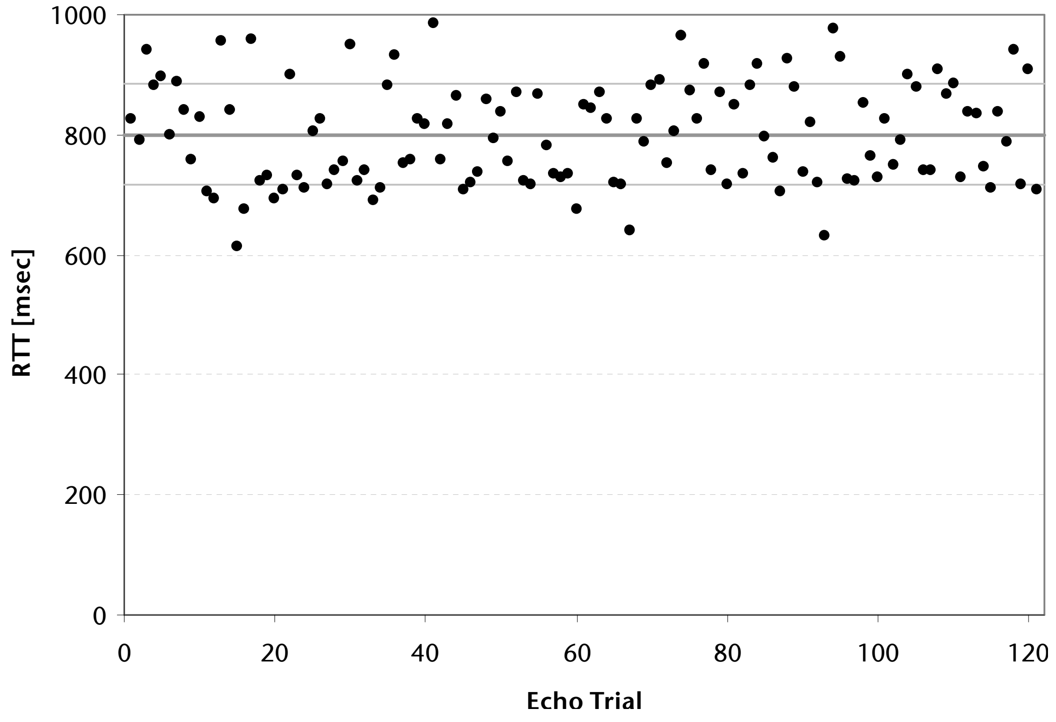

Figure 6.14: Tor round-trip times for 4096 byte messages: In contrast to STONe's random routing the RTTs are more concentrated around the average. The vertical lines mark the average and the standard deviation.

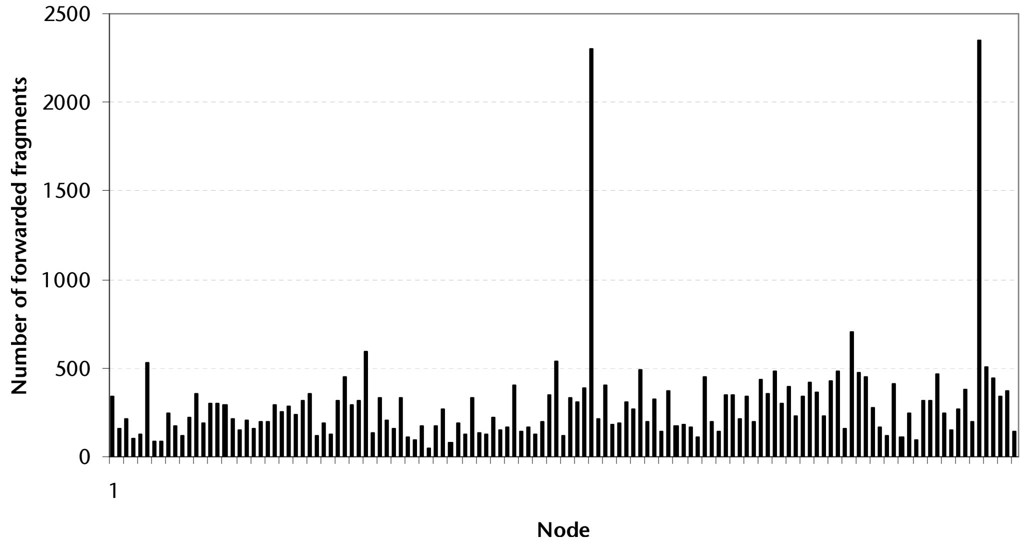

Figure 6.15: Anonymity in Random Routing – Counting the number of forwarded fragments: The routing path is not recognizable, but the sender and receiver clearly stick out of all the nodes in the STONe network.

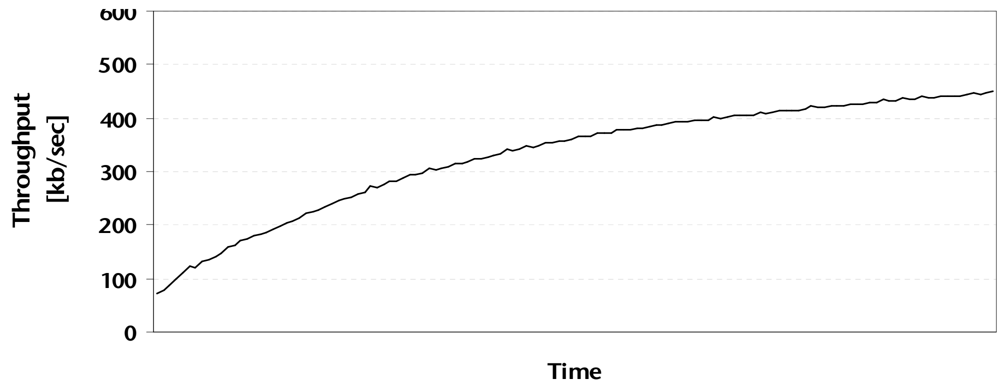

Figure 6.16: Synchronized Communication for $N = 8$: The throughput approaches the average logarithmically when a single connection sends at maximum throughput.

Chapter 7

Related Work

STONe intersects with many different research areas. The first part of this chapter is about Trusted Computing and Trusted Operating Systems. There is a large body of work on these topics, which are long-standing research problems. In the second part of the chapter we survey Secure Communication. In the third part we look into alternative approaches for anonymous communication and Traffic Analysis. After this we compare existing architectures for secure and anonymous communications. In the last part we describe the related work of the applications we implemented on STONe.

7.1 Trusted Computing and Trusted Operating Systems

7.1.1 Trusted Computing

Trusted Computing covers two main areas: One is how to protect execution against an adversary, and the other is how to prove genuity and trust of a platform.

The idea of protecting program execution against tampering in commodity computers is not new [116]. First approaches have been implemented in tamper-resistant modules using cryptographic techniques to secure communication and storage [109]. This work presents the base of today's Trusted Computing systems. Further development are cryptographic Co-processors [201] that implement copy protection, electronic cash or secure postage in distributed systems. Secure cryptographic Co-processors were one of the first systems that provided sealed tamper-resistant storage. Recent research describes how to actually build a sealed storage system for secrets with minimal architectural support [118]. In this research user secrets are decoupled from hardware devices, without the requirement for built-in device secrets.

Execute-only memory [186] is a hardware platform on which memory content is tamper-resistant but not hidden as in Trusted Computing. However, XOM is prone to replay attacks which need to be fixed by additional memory integrity checks [169]. Further studies explore how to run an untrusted operating system on such an architecture [122]. Trusted operating systems have to be evaluated yet. A similar, but different approach to tamper-resistance is AEGIS, a tamper-evident architecture [182]. Similar to XOM, all components external to the processor are untrusted, but XOM provides a larger number of processes and is more flexible on the application-level. Intel's Execute Disable Bit uses a similar concept on the current Intel Pentium architecture to provide a bit that disables code execution in a memory segment. This is a powerful protection method – especially against buffer overruns.

The idea of using virtual machines for strong isolation has been around for a while [133]. Another approach for process isolation in Trusted Computing is to use Virtual Machines [87]. However, for example, by using virtual machines such as the VServer in PlanetLab [11] for isolation and attestation it is possible to implement Trusted Overlay Networks. Microsoft's NGSCB is an implementation for Trusted Computing [79] that uses a similar memory protection scheme. Software-only protection against tampering is a hard problem, but there has been some recent work on control-flow integrity checks against a limited adversary [26].

Authentication and trust on the hardware platform have also been widely studied. Intel was the first to introduce serial numbers for its processors to identify hardware platforms [8]. The secure boot mechanism by Arbaugh [32] has defined a new primitive: *Attestation*. The idea of attestation is that all components of a PC have to be loaded and verified successively, starting with a verifiable small code base as the root of trust – in this case a PC's BIOS. Attestation defines the trust relationship between the components and is also a method for implementing access control. A process is trusted only if it attests to the operating system. The Terra system suggests ways to implement attestation using Virtual Machines and SSL [87], but the downside of their approach is heavy overhead and its limitation to monolithic operating systems. SWATT is a software-only approach to attestation of memory contents [166].

TCPA is an industry consortium that set up a standard for Trusted Computing hardware [21]. In Trusted Computing it is the certified platform key that is a quasi-identification of platforms. However, this raises privacy concerns for attestation since these identities should only be revealed when a platform is compromised. Group signatures and direct anonymous attestation address this problem [51, 48]. There is also research work on trying to identify a platform based on its hardware properties, which could be regarded as an implicit form of authentication. Timing behavior of TLB accesses is a property that is hard to emulate, and it can therefore be used to verify that a particular software runs on the hardware and not a virtual machine [107]. Remote device fingerprinting identifies devices in a network based on the clock skew in TCP timestamps [114].

Trust and security in operating systems is mostly an unsolved problem. Every day new security breaches make the news. Common examples are exploits like buffer overruns or new phishing attacks by malware on the PC [3]. Some worms or viruses are harmless and simply waste bandwidth by spreading themselves through the Internet. Others let the adversary control the victim's system, erase valuable data, and turn compromised systems into a Bot-Net which the adversary can use to attack servers with large resources [103].

The main problem is insecure user interfaces and software bugs that allow system compromises. For example, in some cases a user connects to a server, and he confirms that he wants to see the website despite problems with the certificate. Thereby confidential information may leak or untrusted code may compromise the system.

7.1.2 Trusted Operating Systems

Operating systems need to provide support for applications to protect against these attacks. As functionality of operating systems increases they incorporate more code, but single modules do not get protected. One example is the integration of Internet Explorer into Windows, which over the years has proven not to result in any strong benefits for the user and furthermore opens additional doors for an adversary to attack the OS. Another example is device drivers in operating systems: Device driver crashes become a serious problem for operating systems reliability [184], and virtual machines are an efficient protection method against it. However, being a source for crashes also means being a source for potential security bugs.

The idea for trusted operating systems has been around for decades, but often trust is associated with information flow or access control (e.g.[132]). In the literature, "trust" in operating systems is often not well-understood. Even the original Bell and LaPadula paper on security in Multics [39] does not define what it means for a process to be trusted. Later, Neumann et al. implemented a provably secure operating system (PSOS) [135]. PSOS is a hierarchically-structured capability-based operating system design. Every layer manages objects of a certain type and these objects are accessed by capabilities. PSOS can be considered as the first type-safe operating system. Rushby evaluates approaches for the design and verification of a secure system [155]: He says that security systems should be conceived as distributed systems in which security is achieved by isolation but also partly by trusted functions performed by some system components. The Fluke OS is a first step towards this type of operating system [82].

Trusted Overlay Networks inherently implement end-to-end security in distributed systems [156] and also tolerate some untrusted platforms and give them trust against tampering by using common techniques against Byzantine failures [115].

In general there are three methods for isolating applications, and they all have different root of trusts. First, application-level sandboxing isolates applications from each other [188]. This is similar to virtual machines or simulators on the application-level (e.g. Java, CLR, SimOS, Wine or SoftWindows). Second, virtual memory protects the memory of single processes from each other, and only kernel-level processes can break this protection. Third, hypervisors replicate the hardware as a whole machine; protection depends only on a small hypervisor code base. The problem of the first two approaches is that they cannot protect against operating system faults which is a common cause of failures, and when they communicate with untrusted operating system code additional protection is required [192]. Furthermore, they rely on a large code base for the root of trust.

Earlier papers on hypervisor-based fault-tolerance emphasize the ability for crash recovery [50], and this as well improves the reliability of operating systems against failures and also Byzantine faults. For example, when an adversary gets control over the print spooler because of a software bug he should not automatically get access to the network stack. Isolation and modularization are not necessary between all processes. The most significant protection boundary is between user- and kernel-space, as well as between kernel modules like device drivers. Often an outside adversary exploits a vulnerability in a system call to get access to a root shell and therefore full control over the victim system. With isolation this is not a problem. In a multi-user server system like PlanetLab [38], where every user has her own environment, process isolation is required on the virtual machine level. However, on a single-user system as a PC a local user with root access has control over the whole machine. The main adversary in this scenario is an intruder from the network. Virtual memory protection is good enough for many process-to-process isolation techniques, and the penalty for context-switching is often lower than for virtual machines, since a process does not require suspending and resuming full operating systems state. For high reliability it is useful to have virtual machines, because it is straightforward to reinstate machine state in case of a failure.

A trusted operating system layer is important for clients and servers. On servers it efficiently increases fault-tolerance and robustness against security bugs between different users. On the client-side strong protection has to take place between user-space and kernel-space. Furthermore, user-interfaces require protection as well. The philosophy behind client-side operating systems is that the platform should be open and the user controls everything [117, 79]. Client operating systems have to preserve openness but make administration and user interaction more secure. Thin clients in contrast are centrally controlled and have their applications in local networks [161].

These approaches have never evolved in the wide area, since hardware is still cheaper than network bandwidth. Furthermore, availability of high bandwidth wireless networks is often still restricted outside office buildings, and it is not clear whether a user wants his information stored on a central server that may fail or become insecure. There has to be a trust relationship to this central authority for security and privacy which may only work in local networks, excluding ubiquitous laptop computers.

Virtualizing a whole operating system uses lots of system resources and is hardly scalable on a normal PC [87]. However, new hypervisor approaches that use paravitualization can have much higher performance on commodity operating systems [36] or provide much better scalability [195].

On the application-level in Distributed Systems there has also been a considerate amount of work on classifying information. In program partitioning a compiler partitions a program depending on the trust level of data during computation to protect confidential information from untrusted hosts [202]. Similarly, we can partition programs for privilege separation, where privileged instructions such as *setuid* can be executed in a protected monitor process [52].

System updates and attestation are another crucial aspect of trusted operating systems. Attestation is an operation that verifies authenticity of the code. However, this signature only tells the attesting entity that it is certified by some manufacturer. It does not necessarily verify what it does or which bugs it fixes. Semantic attestation is a new way of defining attestation [93].

A modular architecture that allows trusted extensions similar to Exokernel [80] has advantages for systems updates, since single modules can be easily certified and updated separately. A trusted compiler used in systems like SPIN can support the implementation of these update systems [40].

Microsoft's singularity kernel [98] follows the language-based approach to trusted operating systems by using managed or trusted code in the operating system modules. However, this restricts the software-hardware interface. The operating system vendor controls the virtual machine, and this has the danger that the system becomes a proprietary virtual machine.

7.2 Secure Communication

The most important systems and protocols for secure communication in the Internet are SSH [14], SSL [15], IPsec [108], and TLS [73]. However, even securing communication on the network or application layer does not mean that the network is really secure. The strongest threats are Byzantine failures from malicious attacks.

Byzantine faults in the form of software bugs are common in today's Internet. Nodes get compromised and adversaries take them over to join the node to a whole Bot-Net of compromised nodes. Protecting against Byzantine failures is desirable in distributed systems.

Anonymity protocols require protection of the network stack against Byzantine Faults as well, or it becomes hard to design an anonymity protocol that is reliable against Traffic Analysis. There are already systems that protect against Byzantine faults in networks and distributed applications like secure routing [56] or virtual machine and replication techniques like BFT [57]. It is the idea to use these in anonymity networks as well.

The first alternative to protect the network from Byzantine failures is secure routing systems. They have multiple components: First, they *maintain routing state* against an adversary who is tampering with the system. Second, they *forward messages securely* and protect against malicious routers in the system who eavesdrop on traffic, drop packets, or reroute packets in the network. For example, secure BGP [97] protects against an attacker who tries to modify routing state in BGP, and routing in Fatih [128] protects against an attacker on message forwarding. Another approach for secure routing in structured peer-to-peer networks is to implement self-certification of application data in the network [56] which can tolerate about 25% malicious nodes. An additional problem in peer-to-peer networks is the Sybil attack [78]. It is often easy to obtain network addresses or nodeIDs, and certificates have to be used. This is not necessary for IP addresses, since they are generally more difficult to obtain in large amounts. Another technique is, like in STONe, to use Trusted Computing for distributing node addresses in peer-to-peer networks to protect against these attacks [35].

The second alternative is to protect the network stack of the router against attacks on its state. One possibility is to use techniques from distributed systems such as state machine replication like BFT [57]. However, replication is relatively expensive. In addition it is also possible to establish local protection like standard sandboxing or virtualization techniques to detect malicious behavior or intrusion directly on the router. STONe provides the novel idea to use Trusted Computing to protect the network stack against failures.

Decoupling Byzantine faults from anonymity protocols makes anonymity protocol design easier. When no Byzantine faults occur, intersection attacks are still possible while routers from a single trust domain collude. A single trust domain would be equivalent to a centralized message forwarding network, and this is prone to passive logging attacks. With at least two trust domains, one domain knows the sender and the other the receiver, but neither both. In a system that has random Byzantine faults, a single adversary could control a Bot-Net across trusted domains and protection becomes much more difficult.

In systems with multiple but static trusted domains it is good enough to have a fixed number of multiple hops that are spreading across at least two trust domains that do not collude [84]. In a system with Byzantine faults, however, the path length depends on the total number of nodes in the network, the largest domain size, and the expected fraction of infected hosts that could potentially become part of a Bot-Net. There is a trade-off between implementing a system that protects against Byzantine failures and one that uses longer paths instead.

Byzantine failures also have an adverse effect on the protocol that distributes the routing information. Flooding or broadcasting this information is more robust against Byzantine failures [142], but it is inherently inefficient, since it consumes network bandwidth and router resources that are exponential in the number of messages. When the system does not use flooding, it has to cope with possible traitors that imitate legitimate nodes. In a system without Trusted Computing additional cryptography is needed to protect against these attacks [142].

Lastly, there are Denial of Service attacks that disrupt communication. Distributed Denial of Service attacks (DDoS) can occur on different layers. The brute-force method is to flood a server with network packets to disrupt service. However, this attack requires a vast amount of network bandwidth that is hard to obtain. On the network layer an adversary can exploit a leak in TCP that keeps track of half-open connections in memory. When the adversary floods the server with such SYN requests it runs out of memory for new connections. A common technique is to use SYN cookies to protect against this attack [20]. A SYN cookie contains a unique sequence number that is the keyed hash of the connection information (source IP address and port and destination IP address and port). The server does not have to store information anymore because the packet with the sequence number during the handshake is self-verifying. Furthermore, there exist DDoS attacks on specific protocols [70].

Other methods to protect against DDoS attacks on the network layer are IP traceback and hashback techniques [160, 176], and also the use of capabilities [200]. Because in network layer attacks IP addresses are often mimicked we can modify the routers to add some extra information to the packet that identifies the communication path and therefore the sender. All packets with the same path identification information that have been malicious can be filtered accordingly. These methods can also be used to filter packets.

DDoS attacks are most powerful on the application-level. When an adversary launches an application-level DDoS attack on a server its goal is to starve its internal resources such as CPU or disk I/O. There are multiple ways to protect against this attack. One is to use computational puzzles to decrease the rate of server accesses [71, 194]. Another method is to use reverse Turing tests to distinguish human users from automated attackers [103]. To launch such an attack a whole network of clients is required that simulates a flash crowd. This situation normally only happens when many users access the same website and exceed the normal usage level.

7.3 Anonymous Communication

There exist quite a few commercial or open-source systems for anonymous communication. Anonymizer [2] is a simple trusted HTTP proxy for anonymous web browsing.

It is like a NAT between the sender and the receiver of the connection and protects against passive logging attacks on the receiver by replacing the sender's IP address. However, such systems are a single point of failure and trust, and a closed, non-verifiable system can potentially log and store all source and destination addresses of messages it forwards. Tor is another system for anonymizing communication [75]. In contrast to Anonymizer it uses a set of proxies, and if one proxy gets compromised it does not compromise the whole system. Freenet, in contrast, is a distributed system for file-sharing that provides some anonymity, but its main goal is to be censorship-resistant and to provide anonymity of content. A single file is split up in small pieces that are hard to assign to the originator. Distributed Hashing is used to find all the pieces and put them together [62]. A single file can only be retrieved sequentially, piece by piece.

Today's systems like Anonymizer, Tor, or Freenet are the state-of-the-art for anonymity in distributed systems. However, their protection against Traffic Analysis is often poor. Freenet's protection against Traffic Analysis is based on mix networks, and Tor does not protect against Traffic Analysis at all.

Anonymous communication is a combination of many different areas. One area is steganography. Two parties may hide their communication in some covert channels of network protocol headers or digital images. For example, TCP contains some covert channels [172]. However, most of these techniques have been proven to be insecure.

Theoretically, anonymous communication is related to secure multiparty computation where N parties have private inputs, but they want to compute a boolean circuit that outputs a single public value. An adversary with access to the circuit is not able to tell what the inputs were and who gave which input [89]. Anonymous Communication is only a subset of secure multiparty computation. Using secure multiparty computation for anonymous communication is inherently inefficient [146].

In synchronous networks anonymous communication of N parties can be achieved in $O(\log N)$ steps by rapid mixing [146, 47]. Other systems have been proposed based on secret sharing [63]. However, the asynchronous case which is more common, requires additional cover traffic to hide access patterns. This has been elaborated in the Oblivious RAM [90] where memory access patterns are hidden. Oblivious RAM uses a Batcher network to shuffle the memory locations. This works well in a physical memory because it has a fixed size. In STONe, however, the number of participating nodes is dynamic.

Therefore, Chaum proposed mix networks for asynchronous email communication [58]. Mixes have a variety of applications, especially in voting. A single mix collects messages and dispatches them at random, so that it is hard for an adversary to correlate incoming with outgoing messages. A single mix is a single point of failure when compromised and often also prone to brute-force disclosure attacks [112].

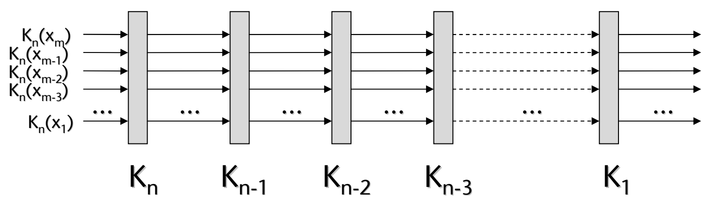

Figure 7.1: Mix Cascades: Several mixes are arranged in a linear chain. Every mix delays messages, and an adversary who only observes one mix is not able to trace messages.

Therefore, mixes are often connected into mix cascades as shown in figure 7.1. The number of hops in a mix cascade depends on the number of different trust domains. Ideally, every hop is within different trust domains. The disadvantage of mix cascades is reliability. Because a cascade is only a fixed path through the network, a single broken mix destroys the communication path. Free peer-to-peer mix networks would solve this problem. However, the degree of anonymity decreases because different paths get routed through different mixes. STONe's random routing alleviates this problem because it changes the path frequently and would therefore let random paths go through the same mixes.

Pool mixes [64] and stop-and-go mixes [112] are the typical type of mixes. Both are used for email communication. The difference is that pool mixes wait until enough messages arrive before they dispatch them, and stop-and-go mixes delay messages randomly. However, stop-and-go mixes are prone to statistical disclosure attacks [66]. When the adversary wants to trace a single message she collects statistical information about all incoming and outgoing messages in the mix. Mix networks known as so-called anonymous remailers have been successfully implemented for email anonymity. Early examples are Babel [91] and Mixmaster [129]. Mixminion is a recent anonymous remailer that now also uses link encryption [69].

In general, mix networks are not useful for low-latency communication. On a busy router there is usually enough traffic for mixing, and additional non-uniform delay can cause packet reordering – a common problem in transport-layer protocols such as TCP. A less efficient way to provide anonymity on a low-latency network is to pad all communication with cover traffic [65, 144].

Many protocols for anonymous low-latency communication therefore only provide sender/receiver anonymity by hiding the sender and receiver addresses [185, 148]. If they do not send cover traffic anonymity depends on the amount of real background traffic. Jap [9] or MorphMix [151] are a systems with mixes that depend on the amount of background traffic, because mixes can only delay packets for a certain time interval, even if there is nothing to shuffle with. Then timing attacks are usually possible [121]. In STONe we pick a different approach because Trusted Overlay Networks already protect the sender address. STONe protects against Traffic Analysis by random routing and synchrony.

There are several reliability and reputation issues in mixes. How can a mix be trusted ? Several issues similar to the ones in STONe come up in mixes as well [74], but if they are trusted we could as well use random routing instead to protect against traffic analysis as shown in this thesis.

Fragile Mixes [149] provide a novel protection mechanism against a mix administrator who is giving away logs. By doing so she would automatically compromise her own anonymity. The assumption is that a mix administrator also participates in anonymous communication. However, this does not protect against an outside adversary eavesdropping on messages.

One of the largest-scaling systems for anonymous routing today is Tor [75]. Tor is based on the idea of onion routing [185] where packets get encrypted in layers and every hop along the route strips off one layer and sees the packet header that contains the address to the next hop. Tor is a large-scale implementation of onion routing and solves problems from the first generation. For example, onion routing was fairly inefficient because it used public key cryptography. Tor now sets up shared keys between the sender and the hops along the path. In addition Tor implements location-hidden services, which are similar to anonymous STONe Sockets. In location-hidden services, two parties – Alice and Bob – trust that a rendez-vouz point will never leak information about them. The rendezvous point is the anonymous address that Alice and Bob use to communicate. However, this is still a single point of trust, and STONe strengthens this vulnerability, because it can rely on the Trusted Computing hardware. Another disadvantage of Tor is that it uses source routing because it needs to compute the onion in advance, and it is less flexible in resilience than STONe.

Cashmere or P5 address resilience in anonymous communication by using group communication [204, 170]. Every group in the network shares the same key, and encrypted packets get forwarded to all members of a group. Intersection attacks are the main problem of these protocols. By leaving a group and joining another an adversary can learn which nodes are online, and she can also decrypt traffic of this group. In STONe we decided to solve the resilience problem by adding redundancy to the overlay structure and not by using multicast. STONe is equivalent to a multicast network with N groups where N is the number of nodes. Mix networks are known to be vulnerable to timing attacks [121].

Crowds [148] uses a different approach for anonymous communication. In Crowds, upon message arrival anode flips a coin and either forwards the message randomly to a hop or sends it directly to the receiver. This technique is similar to random routing in STONe, but the problem in Crowds is that it does not provide receiver anonymity. Also, it is basically an anonymous web proxy and does not care about application-level communication. Frequent path reformations make most anonymity systems, especially Crowds, vulnerable against the Predecessor attack in which an adversary finds the sender by investigating the predecessor of the message [171].

Tarzan [84] is a peer-to-peer overlay network that is transparent on the IP-layer. It sets up circuits in advance and uses local mimic traffic to hide traffic patterns. In STONe we decided in favor of a global traffic scheme that may increase the latency but provides stronger protection against anonymity. Tarzan also addresses the issue of application transparency on IP-Level, but does not integrate application-level anonymity with anonymous routing like STONe. The link from Tarzan to the application is a vulnerability. STONe also has the advantage that it gets strong hardware support and does not have to cope with malicious hosts. Tarzan also uses source routing and cannot easily route around failures.

Broadcast networks like DC-Nets [59, 175] or XOR-Trees [77] are synchronous. In Dining Cryptographers every node broadcasts an encrypted message at the same time to all other parties, and the parties are not able to trace the originator of the message. In DC-Nets, however, all parties have to play fair because they can jam the communication. The advantage of DC-Nets is that there is no additional delay in the communication, but instead they use bandwidth. XOR-Trees implement a broadcast tree to reduce the number of messages in the broadcast system.

There are controversies about the ethics of anonymous communication [44, 137]. People can use the communication networks to establish “Darknets” to exchange illegal content. Digital communication networks enhance old-fashioned “sneaker nets” to exchange pirated software. This, of course, comes down to ongoing legal disputes between content providers and network providers as in the MGM vs. Grokster lawsuit [12].

System	Routing	Traffic Analysis	App
Anonymizer [2]	Single Proxy	Trusted Proxy	Yes
Freenet [62]	Onion Routing	Mix Network	No
Freedom [33]	Onion Routing	Random Walk	Yes
Pipenet [65]	Onion Routing	Cover Traffic	No
Tor [75]	Onion Routing	Random Walk	Yes
Crowds [148]	IP	Random Walk	No
Jap [9]	Onion Routing	Mix Network	No
Herbivore [175]	DC	Broadcast	No
Cashmere [204]	Onion Routing	Broadcast	Yes
P5 [170]	IP	Broadcast	No
Tarzan [84]	Onion Routing	Cover Traffic	Yes
MorphMix [151]	Onion Routing	Mix Network	No
Anon [96]	Onion Routing	Cover Traffic	Yes
ISDNMixes [144]	Switched Circuit	Mix Network	Yes

Table 7.1: Comparison of Systems for Anonymous Communication

Table 7.1 gives an overview of common systems for anonymous communication. The second column contains the type of anonymous routing the system uses. A singe proxy means that the system only consists of one large proxy network that is firewalled from the outside. Another alternative is just plain Internet routing (IP), Overlay routing, Dining Cryptographer's (DC) or Onion Routing. The next column shows the measure the system provides against Traffic Analysis. A trusted proxy means that the anonymous routing system is trusted and shut off from the attacker. It may or may not provide measures against end-to-end traffic analysis. In a random walk the client picks a set of random nodes to form the path. A broadcast scheme protects against Traffic Analysis by sending the same message to multiple nodes. Nodes may also send cover traffic only to protect against Traffic Analysis. Mix networks are also commonly used. They sometimes include a random walk and cover traffic, but the basic characteristic of a mix is that it delays and shuffles messages. The fourth column describes whether the system supports application-level anonymity. Often the systems use pseudonyms instead of IP addresses and create hidden rendez-vous point like Tor.

7.4 Overlay Networks and Internet Architectures

7.4.1 Overlay Networks

Modern overlay networks emerged almost a decade ago with the advent of Internet services. The main problem was to enhance performance for web browsing, and therefore some kind of web caching method had to be established. The theoretical base of this work for structured overlays is consistent hashing [105], from which the Akamai network [1] emerged. Today there exists a large variety of overlay networks – structured or unstructured.

STONe borrows several ideas from structured overlay networks and uses a hypercube topology similar to CAN [147] to enhance load-balancing and resilience. The difference between STONe and content-distribution networks [158] is that STONe is neither a location service that finds objects in distributed systems nor a distributed storage system. STONe a routing overlay. It does not have to optimize for caching performance or replicas, but it needs to provide for alternate routes.

Topologies for content-distribution networks other than CAN include a ring [180] or tree [154] with different failure properties. The main conclusion of that research is that a ring has the best fault-tolerance properties [92]. Bamboo [152] is a re-engineered DHT for Pastry that optimizes for frequent and large membership changes.

Two popular wide-area implementations of these suggested networks are CoDeeN [193], Tapestry [203], and OpenDHT [153]. CoDeeN is a network of proxy servers for content-distribution, whereas OpenDHT is a distributed storage facility. Tapestry [203] implements a routing overlay for locating objects and services.

The downside of structured overlays is that they do not adapt well to the heterogeneity of the Internet. They also have high maintenance costs and are limited in searching for data, since they only support simple exact-match. Therefore, standard file-sharing applications often use unstructured overlays [7, 10]. Some of these problems, like searching, are more of a problem in content-distribution overlays than in routing overlays. However, most of these problems in structured overlays can be fixed with hybrid properties of unstructured overlay [55], and STONe also uses some of these results to optimize performance.

7.4.2 Internet Architectures

Overlay networks have the ability to fix problems in existing Internet architectures or extend them with new features. For example, Resilient Overlay Networks (RON) [30] have been designed to alleviate common problems of path failures in the Internet [159]. Another category is Internet security architectures that protect against Denial of Service attacks [29, 110, 179, 200]. DOA is an overlay that extends the Internet address space beyond NATs [191]. Other uses of overlays extend existing Internet functionality like multicast [101] and QoS [181]. Anonymity networks like Tor can also be seen as an Internet anonymity architecture [75]. In STONe participating nodes are authenticated by Trusted Computing hardware which strengthens the security and anonymity, especially against Byzantine failures. SOS and Mayday only use lightweight authentication and have weaker mechanisms against Denial of Service attacks.

In addition to overlays that extend the Internet architecture there exist several approaches for a next-generation Internet routing architecture. The Nimrod routing architecture uses network maps (like road maps) instead of routing tables and lets the clients pick the routes, i.e. uses source routing [54]. Nimrod was designed to make the network scalable to a large number of nodes. Recent advances on next-generation Internet architectures are NIRA [199] and FARA [61]. They try to overcome the addressing problem in the Internet and also address the lack of resilience.

7.5 Instant Messaging

There exist a variety of Instant Messaging systems that are centralized such as AIM [4], Windows Messenger [24] or Yahoo! Messenger [25]. Some of them have questionable privacy policies that allow them to record messages arbitrarily on the central server (e.g. [4]). There are some new system that provide end-to-end privacy and security using public key encryption [23]. This protects privacy in a centralized system but does not protect against Traffic Analysis. Skype as a Voice-over-IP system is also a distributed approach to Instant Messaging [37]. Similar to STONe, it uses an overlay structure for forwarding messages reliably, but it also does not provide protection against Traffic Analysis.

7.6 Filesystems and File Sharing Networks

There is a large number of network file systems for local- or wide-area networks that optimize file system performance by caching, as for example NFS [157], AFS [95], and xFS [31]. The Coda file system replicates data on multiple servers to improve availability [113].

SFS was one of the first file systems that explicitly provides server certification [124]. When a client accesses a file it finds the public key in the file name and uses this public key to access the server. This decouples key management from the file system and prevents an adversary from tampering with file names. The SiRiUS file system [88] provides a security layer file systems even without a trusted server for access control as in SFS. SiRiUS uses cryptography to provide access control. Farsite [27] is a distributed decentralized secure file system that protects against Byzantine faults in an untrusted environment, but it does not protect against Traffic Analysis.

In addition to standard network file systems there exists a variety of file sharing systems that are design for publish-subscribe operations. These publish-subscribe systems are resistant against censorship and protect privacy. Common examples are Publius [190], Tangler [189], Freenet [62], BitTorrent [5], and Mnemosyn [94]. A recent study has shown that most of these implementations hide the traffic [104] against simple mimic attacks. However, we know only that Freenet uses mixes to protect against Traffic Analysis.

Chapter 8

Conclusion and Future Work

We have presented the design and implementation of Secure and Trusted Overlay Networks (STONe). STONe demonstrates that emerging Trusted Computing technologies would provide a much better platform for anonymous communication compared to today's anonymity systems. STONe's security is based on two cornerstones: A hardware-based Trusted Computing Platform and an additional secret key. A user can only enter the system to send and receive messages anonymously if he is in possession of both. If a traitor gets detected its TCB comes on the blacklist and is hence excluded from the system. To compromise the system an adversary has to overcome the cost of purchasing Trusted Computing Hardware and know the changing secret key. This model is much stronger than in current systems for anonymous communication, but it is realistic and finally helps to implement anonymous communication. This work explains the issues and pitfalls that occur when designing a more efficient system for anonymous communication based on Trusted Computing.

We designed STONe as an overlay network that uses Trusted Computing to isolate Byzantine failures, which makes it possible to decouple protection against traffic analysis from network routing, thus providing more efficient and more secure anonymous communication. STONe is resilient against churn and congestion, even in a large-scale environment, but it also provides strong protection against traffic analysis. To achieve these goals STONe uses random routing over a regular network topology such as a hypercube, a novel technique for anonymous routing. Unlike mix networks for anonymous communication random routing over such technologies is self-mixing and does not require explicit message shuffling.

At the application-level, STONe provides a socket endpoint to access the anonymity network and a trusted name service that maps names to self-certifying anonymous identities. This delivers anonymity to the application endpoint and makes anonymous communication more robust against attacks that target application behavior, such as name server queries. Further, it prevents a malicious application from mimicking an arbitrary identity. We have built two applications on top of STONe–Anonymous Instant Messaging and an Anonymous File System. The results of our experiments verify our claims.

STONe can have many applications, and is not only useful on the Internet. Embedded devices that use smartcards are protected against any outside attackers, and STONe can provide anonymity and therefore enhanced security as well. An example is wireless communication of aircraft components or car security.

There are many opportunities for future work on STONe. We still have to evaluate STONe in a real system with Trusted Computing hardware and operating systems that meet the requirements of Trusted Overlay Network. It is crucial that remote attestation protocols are able to detect any compromised node and only admit a negligible number of false negatives to the network. TCBs have to be verifiable to prevent backdoors, and the keys have to be protected by additional hardware and software tamper-resistance measures. More research on the robustness and security of these methods is definitely needed.

It is also an open problem how to efficiently combine anonymity protocols for untrusted systems like Onion Routing with protocols for trusted systems like Random Routing. It is also desirable to implement a distributed trusted name server. When a single node with all name entries entries leaves the network, STONe has to be able to restore the membership list. Lastly, the fragment size is an open issue. It is unclear whether it is necessary to use uniform fragment sizes or vary the size.

Bibliography

[1] Akamai. http://www.akamai.com.

[2] Anonymizer.com. http://www.anonymizer.com.

[3] Anti-Phishing Working Group. http://www.antiphishing.org.

[4] Aol instant messenger. http://www.aim.com.

[5] Bittorrent. http://www.bitrorrent.com.

[6] Epic. http://www.epic.org.

[7] Gnutella. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[8] Intel serial numbers processors to secure software and internet. http://www.electronicsweekly.com/2005/03/18/technology/personaltech/scamArticle14335.html.

[9] Jap anonymity & privacy. http://anon.inf.tu-dresden.de.

[10] Kazaa. http://www.kazaa.com.

[11] Linux VServer. http://linux-vserver.org.

[12] Mgm vs. grokster. http://www.eff.org/IP/P2P/MGM_v_Grokster.

[13] Network general. http://www.networkgeneral.com.

[14] Openssh. http://www.openssh.org.

[15] Openssl. http://www.openssl.org.

[16] Phishing iq test. http://www.mailfrontier.com/forms/msft_iq_test.html.

[17] Planetlab all pairs pings. http://pdos.csail.mit.edu/s̃trib/pl_app/.

[18] Privoxy. http://www.privoxy.org.

[19] Seti@home. http://setiathome.ssl.berkeley.edu.

[20] Syn cookies. http://cr.yp.to/syncookies.html.

[21] Trusted computing platform alliance. http://www.trustedpc.org.

[22] Verisign. http://www.verisign.com.

[23] Voltage security. http://www.voltage.com.

[24] Windows messenger. http://www.microsoft.com/windows/messenger.

[25] Yahoo! messenger. http://pager.yahoo.com.

[26] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *CCS 2005*.

[27] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI 2002*.

[28] AMD Corporation. Pacifica - Next Generation Architecture for Efficient Virtual Machines. http://developer.amd.com/assets/WinHEC2005_Pacifica_Virtualization.pdf.

[29] D. Andersen. Mayday: Distributed filtering for internet services. In *USITS 2003*.

[30] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP 2001*.

[31] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *SOSP 1995*.

[32] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *IEEE Symp. on Sec. and Priv.*, pages 65–71, 1997.

[33] A. Back, I. Goldberg, and A. Shostack. Freedom systems 2.1 security issues and analysis, 2001.

[34] A. Back, U. Möller, and A. Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. *LNCS*, 2137.

[35] S. Balfe, A. Lakhani, and K. Paterson. *Trusted Computing*, chapter Securing Peer-to-Peer networks using Trusted Computing. IEE Press, 2005.

[36] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, pages 164–177, 2003.

[37] S. A. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. Technical Report CUCS-039-04, Columbia University, 2004.

[38] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *NSDI*, pages 253–266, 2004.

[39] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical report, The MITRE Corporation, 1976.

[40] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *SOSP*, pages 267–283, 1995.

[41] O. Berthold, A. Pfitzmann, and R. Standtke. The disadvantage of free MIX routes and how to overcome them. *LNCS*, 2009, 2000.

[42] J. Bethencourt, J. Franklin, and M. Vernon. Mapping internet sensors with probe response attacks. In *Usenix Security 2005*.

[43] D. Bickson and D. Malkhi. Privacy degradation in the gnutella network. Technical report, The Hebrew University of Jerusalem, 2003.

[44] P. Biddle, P. England, M. Peinado, and B. Willman. The darknet and the future of content protection. *LNCS*, 2696, 2002.

[45] G. R. Blakley. Safeguarding cryptographic keys. In *National Computer Conference*, number 48, pages 313–317, 1979.

[46] E. Blanton and M. Allman. On making tcp more robust to packet reordering. *ACM Computer Communication Review*, 32(1), 2002.

[47] B. Bollobas. *Modern Graph Theory*. Springer, 2002.

[48] D. Boneh and H. Shacham. Group signatures with verifier-local revocation. In *CCS 2004*.

[49] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.

[50] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. In *SOSP*, pages 1–11, 1995.

[51] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *CCS 2004*.

[52] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for priviledge separation. In *Usenix Security 2004*.

[53] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable encryption. *LNCS*, 1264, 1999.

[54] I. Castineyra, N. Chiappa, and M. Steenstrup. RFC 1992 - the nimrod routing architecture.

[55] M. Castro, M. Costa, and A. Rowstron. Debunking some myths about structured and unstructured overlays. In *NSDI 2005*.

[56] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *OSDI 2002*.

[57] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *OSDI*, 1999.

[58] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *CACM*, 24(2):84–88, Feb. 1981.

[59] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *JACM*, (1):65–75, 1988.

[60] Y. Chen, D. Bindel, H. Song, and R. H. Katz. An algebraic approach to practical and scalable overlay network monitoring. In *SIGCOMM 2004*.

[61] D. Clark, R. Braden, A. Falk, and V. Pingali. Fara: Reorganizing the addressing architecture. In *ACM SIGCOMM FDNA 2003*.

[62] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hongang. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *LNCS*, volume 2009, 2001.

[63] D. Cooper and K. Birman. Preserving privacy in network of mobile computers. In *IEEE Security and Privacy 1995*.

[64] L. Cottrell. Mixmaster and remailer attacks, 1994. http://www.obscura.com/ loki/remailer/remailer-essay.html.

[65] W. Dai. Pipenet 1.1, 1996. http://www.eskimo.com/ weidai/pipenet.txt.

[66] G. Danezis. The statistical disclosure attack. In *Sec2003*.

[67] G. Danezis. Mix-networks with restricted routes. *LNCS*, 2760, 2003.

[68] G. Danezis. The traffic analysis of continuous-time mixes. *LNCS*, 3424, 2004.

[69] G. Danezis, R. Dingledine, and N. Matthewson. Mixminion: A next-generation anonymous remailer. In *IEEE Security and Privacy 2003*.

[70] N. Daswani and H. Garcia-Molina. Query-flood dos attacks on gnutella. In *CCS 2002*.

[71] D. Dean and A. Stubblefield. Using client puzzles to protect tls. In *Usenix Security 2001*.

[72] C. Diaz, S. Seys, J. Claessens, and B. Preneel. Towards measuring anonymity. In *LNCS*, volume 2482, 2003.

[73] T. Dierks and C. Allen. RFC 2246: The TLS Protocol, 1999.

[74] R. Dingledine, M. J. Freedman, D. Hopwood, and D. Molnar. A reputation system to increase mix-net reliability. *LNCS*, 2137.

[75] R. Dingledine, N. Matthewson, and P. Syverson. Tor: The second-generation onion router. In *Usenix Security 2004*.

[76] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions of Information Theory*, 2(29):198–208, 1983.

[77] S. Dolev and R. Ostrovsky. Xor-trees for efficient anonymous multicast and reception. *ACM Transactions on Information and System Security*, 3(2):63–84, 2000.

[78] J. Douceur. The Sybil Attack. In *Proceedings of the 1st International Peer To Peer Systems Workshop (IPTPS 2002)*, March 2002.

[79] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. *Computer*, 36(7):55–62, 2003.

[80] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP*, pages 251–266, 1995.

[81] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of rc4. *LNCS*, 2259.

[82] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet Recursive Virtual Machines. *SIGOPS Oper. Syst. Rev.*, 30(SI):137–151, 1996.

[83] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *Usenix Technical 2005*.

[84] M. J. Freedman and R. Morris. Tarzan: a peer-to-peer anonymizing network layer. In *CCS 2002*.

[85] V. Fuller, T. Li, Y. J, and K. Varadhan. RFC 1519 - classless inter-domain routing (cidr): An address assignment and aggregation strategy.

[86] S. Garfinkel. *PGP Pretty Good Privacy*. O'Reilly, 1994.

[87] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP 2003*.

[88] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: securing remote untrusted storage. In *NDSS 2003*.

[89] O. Goldreich. Secure multi-party computation. http://www.wisdom.weizmann.ac.il/ oded/pp.html.

[90] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *JACM*, 43(3):431–473, May 1996.

[91] C. Gulcu and G. Tsudik. Mixing e-mail with babel. In *NDSS 1996*.

[92] K. P. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry n resilience and proximity. In *SIGCOMM 2003*.

[93] V. Haldar, D. Chandra, and M. Franz. Semantic Remote Attestation - Virtual Machine Directed Approach to Trusted Computing. In *Usenix VM*, pages 29–41, 2004.

[94] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In *IPTPS 2002*.

[95] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Science*, 6(1):51–81, 1988.

[96] H.T.Kung, C.-M. Cheng, K.-S. Tan, and S. Bradner. Design and analysis of an ip-layer anonymizing infrastructure. In *IEEE DISCX 2003*.

[97] Y.-C. Hu, A. Perrig, and M. A. Sirbu. Spv: secure path vector routing for securing bgp. In *SIGCOMM 2004*.

[98] G. Hunt, J. Larus, D. Tarditi, and T. Wobber. Broad New OS Research: Challenges and Opportunities. In *HotOS X*, 2005.

[99] Intel Corporation. LaGrande technology.

[100] J. Ioannidis and M. Blaze. The architecture and implementation of network-layer security under unix. In *Usenix Security 1993*.

[101] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O. Jr. Overcast: Reliable multicastig with an overlay network. In *OSDI 2000*.

[102] M. Kaminsky, G. Savvides, D. Mazieres, and M. F. Kasshoek. Decentralized user authentication in a global file system. In *OSDI 2004*.

[103] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds. In *NSDI*, 2005.

[104] T. Karagiannis, A. Broido, N. Brownlee, kc claffy, and M. Faloutsos. Is p2p dying or just hiding ? In *Globecomm 2004*.

[105] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC 1997*.

[106] S. Katzenbeisser and F. A. P. Petitcolas, editors. *Information hiding techniques for steganography and digital watermarking*. Artech House Books, 1999.

[107] R. Kennell and L. Jamieson. Establishing the genuity of remote computer systems. In *Usenix Security 2003*.

[108] S. Kent. RFC 2401: Security Architecture for the Internet Protocol, 1998.

[109] S. T. Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, MIT-LCS, 1980.

[110] A. D. Keromytis, V. Misra, and D. Rubenstein. Sos: Secure overlay services. In *SIGCOMM 2002*.

[111] P. L. Kerstein. How can we stop phishing and pharming scams ? CSO Magazine July 20th, 2005.

[112] D. Kesdogan, J. Egner, and R. Buschkes. Stop-and-go-mixes providing probabilistic anonymity in an open system. In *LNCS*, volume 1525, 1998.

[113] J. Kistler and M. Satyanarayan. Disconnected operation in the code file system. *TOCS*, 10(1), 1992.

[114] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. In *IEEE Security and Privacy 2005*.

[115] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[116] B. W. Lampson. A note on the confinement problem. *CACM*, 16(10), 1973.

[117] B. W. Lampson and R. F. Sproull. An Open Operating System for a Single-User Machine. In *SOSP*, pages 98–105, 1979.

[118] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *ISCA 2005*.

[119] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. RFC 1928 - SOCKS Protocol Version 5.

[120] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1991.

[121] B. N. Levine, M. K. Reiter, C. Wang, and M. K. Wright. Timing attacks in low-latency mix-based systems. In *Proc. of Financial Cryptography (FC'04)*.

[122] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP 2003*.

[123] D. Mazieres and M. Kaashoek. The design, implementation, and operation of an email pseudonym server. In *CCS 1998*.

[124] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP 1999*.

[125] A. D. McDonald and M. G. Kuhn. Stegfs: A steganographic file system for linux. *LNCS*, 1768, 1999.

[126] I. Mironov. Not so perfect shuffles in rc4. *LNCS*, 2442, 2002.

[127] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. Wallach. AP3: a cooperative, decentralized service providing anonymous communication. In *SIGOPS Europe 2004*.

[128] A. T. Mizrak, Y.-C. Cheng, K. Marzullo, and S. Savage. Fatih: Detecting and isolating malicious routers. In *IEEE DSN 2005*.

[129] U. Moeller, L. Cottrell, P. Palfrader, and L. Sassaman. Mixmaster protocol version 2. Internet-Draft, 2005.

[130] J. Moy. RFC 2328: OSPF Version 2, 1998.

[131] S. Murdoch and G. Danezis. Low-cost traffic analysis on tor. In *IEEE Security and Privacy 2005*.

[132] National Security Agency. Security-Enhanced Linux.

[133] National Security Agency. Device for and Method of Secure Computing using Virtual Machines. United States Patent 6,922,774, 2005.

[134] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *PLDI 1998*.

[135] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International Computer Science Laboratory, May 1980.

[136] A. M. Odlyzko. The rise and fall of knapsack cryptosystems. *Cryptology and Computational Number Theory, American Mathemcatical Society, Proc. Symp. Appl. Math.*, (42):75–88, 1990.

[137] V. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The dark side of the web: An open proxy's view. In *HotNets II*.

[138] C. Partridge, D. Cousins, A. W. Jackson, R. Krishnan, T. Saxena, and W. T. Strayer. Using signal processing to analyze wireless data traffic. In *ACM Workshop on Wireless Security*, 2002.

[139] V. Paxson. End-to-end routing behavior in the internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, 1997.

[140] V. Paxson. End-to-end internet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[141] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[142] R. Perlman. *Network Layer Protocol with Byzantine Robustness*. PhD thesis, MIT, 1988.

[143] A. Pfitzmann and M. Köhntopp. Anonymity, unobservability and pseudonymity – a proposal for terminology. *LNCS*, 2009, 2001.

[144] A. Pfitzmann, B. Pfitzmann, and M. Waidner. Isdn-mixes: Untraceable communication with small bandwidth overhead. *Informatik-Fachberichte*, 267:451–463, 1991.

[145] K. Poulsen. Fbi retires carnivore. The Register, Jan 15 2005.

[146] C. Rackoff and D. R. Simon. Cryptographic defense against traffic analysis. In *STOC 1993*.

[147] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM 2001*.

[148] M. K. Reiter and A. D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, Nov. 1998.

[149] M. K. Reiter and X. Wang. Fragile mixing. In *CCS 2004*.

[150] Y. Rekhter and T. Li. RFC 1771: A Border Gateway Protocol 4, 1995.

[151] M. Rennhard and B. Plattner. Introducing morphmix: Peer-to-peer based anonymous internet usage with collusion detection. In *WPES 2002*.

[152] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Usenix Technical 2004*.

[153] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: A public dht service and its uses. In *SIGCOMM 2005*.

[154] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Conf. on Dist. Syst. Platf. (Middleware)*, 2001.

[155] J. M. Rushby. Design and Verification of Secure Systems. In *SOSP*, pages 12–21, 1981.

[156] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[157] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *USENIX 1985*.

[158] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of internet content delivery systems. In *OSDI 2002*.

[159] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: Informed internet routing and transport. *IEEE Micro*, 19(1):50–59, 1999.

[160] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for ip traceback. In *SIGCOMM 2000*.

[161] B. K. Schmidt, M. S. Lam, and J. D. Northcutt. The Interactive Performance of SLIM: A Stateless, Thin-Client Architecture. In *SOSP*, pages 32–47, 1999.

[162] S. Schoen. Trusted computing: Promise and risk. Electronic Frontier Foundation.

[163] A. Serjantov and G. Danezis. Towards an information theoretic metric for anonymity. In *LNCS*, volume 2482, 2003.

[164] A. Serjantov and S. J. Murdoch. Message splitting against the partial adversary. In *PET 2005*.

[165] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP 2005*.

[166] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Security and Privacy 2004*.

[167] A. Shamir. How to share a secret. *CACM*, 22(1):612–613, 1979.

[168] C. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 1948.

[169] W. Shapiro and R. Vingralek. How to manage persistant state in drm systems. *LNCS*, 2320, 2002.

[170] R. Sherwood, B. Bhattacharjee, and A. Srinivasan. P5: A protocol for scalable anonymous communication. In *IEEE Security and Priacy 2002*.

[171] V. Shmatikov. Probabilistic model checking of an anonymity system. *JACM*, 2005.

[172] G. J. Simmons. The prisoners' problem and the subliminal channel. In *CRYPTO 1983*.

[173] Y. G. Sinai. *Probability Theory*. Springer, 1992.

[174] S. Singh. *The Code Book*. Anchor, 2000.

[175] E. G. Sirer, S. Goel, M. Robson, and D. Engin. Eluding Carnivores: File Sharing with Strong Anonymity. In *European SIGOPS Workshop*, 2004.

[176] A. Snoeren, C. Partridge, I. Sanchez, C. Jones, F. Tchakountio, S. Kent, and W. Strayer. Hash-based ip traceback. In *SIGCOMM 2001*.

[177] T. Spalink. *Deterministic Sharing of Distributed Resources*. PhD thesis, Princeton University, 2006.

[178] W. Stevens. RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, January 1997.

[179] I. Stoica, D. Adkins, S. Zhuang, S. Surana, and S. Shenker. Internet indirection infrastructure. In *SIGCOMM 2002*.

[180] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. In *SIGCOMM 2002*.

[181] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. Overqos: An overlay based architecture for enhancing internet qos. In *NSDI 2004*.

[182] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *ISC 2003*.

[183] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu. Statistical identication of encrypted web browsing traffic. In *IEEE Security and Privacy 2002*.

[184] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP*, pages 207–222, 2003.

[185] P. F. Syverson, D. M. Goldschlag, and M. G. Reed. Anonymous connections and onion routing. In *IEEE Security and Privacy 1997*.

[186] C. Thekkath, D. Boneh, D. Lie, J. Mitchell, M. Horowitz, M. Mitchell, and P. Lincoln. Architectural support for copy and tamper resistant software. In *ASPLOS 2000*.

[187] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *STOC 1981*.

[188] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *SOSP*, pages 203–216, 1993.

[189] M. Waldman and D. Mazieres. Tangler: A censorship-resistant publishing system based on document entanglements. In *CCS 2001*.

[190] M. Waldman, A. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant web publishing system. In *Usenix Security 2000*.

[191] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middelboxes no longer considered harmful. In *OSDI 2004*.

[192] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architecture for Java. In *SOSP*, pages 116–128, 1997.

[193] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *Usenix Technical 2004*.

[194] B. R. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for dos resistance. In *CCS 2004*.

[195] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, 2002.

[196] H. S. Wilf. *generatingfunctionology*. Academic Press, 1994.

[197] M. K. Wright, M. Adler, B. N. Levine, and C. Shields. Defending anonymous communication against passive logging attacks. In *IEEE Symposium o Security and Privacy 2003*.

[198] M. K. Wright, M. Adler, B. N. Levine, and C. Shields. The predecessor attack: An analysis of a threat to anonymous communication systems. *ACM Transactions on Information and System Security*, 7(4):489–522, November 2004.

[199] X. Yang. Nira: A new internet routing architecture. In *ACM SIGCOMM FDNA 2003*.

[200] X. Yang, D. Wetherall, and T. Anderson. A dos-limiting network architecture. In *SIGCOMM 2005*.

[201] B. Yee. *Using Secure Coprocessors*. PhD thesis, CMU, 1994.

[202] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Secure program partitioning. In *SOSP 2001*.

[203] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UCB.

[204] L. Zhuang, F. Zhou, B. Y. Zhao, and A. Rowstron. Cashmere: Resilient anonymous routing. In *NSDI 2005*.