BridgeSPA: A Single Packet Authorization System for Tor Bridges

by

Robin Smits

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

Abstract

Tor is a network designed for low-latency anonymous communications. Tor clients form circuits through relays that are listed in a public directory, and then relay their encrypted traffic through these circuits. This indirection makes it difficult for a local adversary to determine with whom a particular Tor user is communicating. Tor may also be used to circumvent regional Internet censorship, since the final hop of a user's connection can be in a different country. In response, some local adversaries restrict access to Tor by blocking each of the publicly listed relays. To deal with such an adversary, Tor uses *bridges*, which are unlisted relays that can be used as alternative entry points into the Tor network. Unfortunately, issues with Tor's bridge implementation make it easy to discover large numbers of bridges. This makes bridges easy to block. Also, an adversary that hoards this information may use it to determine when each bridge is online over time. If a bridge operator also browses with Tor on the same machine, this information may be sufficient to deanonymize him. We present BridgeSPA as a method to mitigate these issues. A client using BridgeSPA relies on innocuous single packet authorization (SPA) to present a time-limited key to a bridge. Before this authorization takes place, the bridge will not reveal whether it is online. We have implemented BridgeSPA as a working proof-of-concept for GNU/Linux systems. The implementation is available under a free licence. We have integrated our implementation to work in an OpenWRT environment. This enables BridgeSPA support for any client behind a deployed BridgeSPA OpenWRT router, no matter which operating system they are running.

Acknowledgements

I thank my supervisor Ian Goldberg for always providing useful feedback. I also thank my co-authors of the project (Divam Jain and Sarah Pidcock) and paper (Divam Jain, Sarah Pidcock, Ian Goldberg and Urs Hengartner) that this work is extended from. I thank Eugene Vasserman for some helpful discussions about his work as well as this work during his visit to Waterloo. Finally, I thank my thesis readers (Srinivasan Keshav and Urs Hengartner) for their time and feedback.

Dedication

I dedicate this thesis to Joey, Ollie, and Suzy.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Tor and Bridges 1

1.2 Bridge Aliveness Attack 4

1.3 BridgeSPA 6

2 Related Work 8

2.1 Comparison with Other Knocking Systems 12

2.1.1 Knockknock 12

2.1.2 Fwknop 13

3 The BridgeSPA Protocol 14

3.1 Adversarial Model 15

3.2 Protocol Details 15

4 BridgeSPA Implementations 19

4.1 Unlisted Bridges 21

4.2 GNU/Linux Standard Implementation 22

4.2.1 Performance Impact 22

4.3 OpenWRT Implementation 23

4.3.1 Implementation Details 24

4.3.2 Performance Impact 25

5 Attacks **29**

5.1 Bridge Aliveness and Enumeration Attacks 29

5.1.1 Other Aliveness Checks 30

5.2 Bridge Client Detection Attacks 30

5.2.1 Active Adversaries 30

5.2.2 Passive Adversaries 33

6 TCP/IP Covert Channels in Microsoft Windows **39**

6.1 Initial Sequence Number Selection 39

6.2 Other Covert Channels 44

7 Future Work **45**

7.1 Kernel Implementation 45

7.2 Properly Handling SYN Retransmits 45

7.3 Properly Handling Finished Connections 46

7.4 Bridge Authority and BridgeDB Changes 46

7.5 Multiple NAT Layers 46

7.6 Pluggable Transports 47

7.7 Privacy and TCP/IP Varieties 47

8 Conclusion **49**

References **50**

List of Tables

2.1 An IP packet header. 11

2.2 A TCP packet header. 11

4.1 Throughput measurements for the KnockProxy (standard implementation). 23

4.2 Throughput measurements for the KnockProxy (OpenWRT implementation), 100 Mb/s. 26

4.3 Throughput measurements for the KnockProxy (OpenWRT implementation), 1536 and 600 KiB/s. 26

5.1 DoorKeeper delay measurement (standard implementation). 34

5.2 DoorKeeper delay measurement in a ModelNet setting (standard implementation). 36

5.3 DoorKeeper delay measurement (OpenWRT implementation). 36

6.1 NIST test suite output, 675000 Microsoft Windows ISNs 42

6.2 NIST test suite output, 675000 GNU/Linux ISNs 43

List of Figures

1.1 A client constructing a 3-node Tor circuit. 2

1.2 A bridge client constructing a 3-node Tor circuit. 3

1.3 Basic interaction among Tor bridge entities. 4

3.1 Changes to Tor bridge-related communication in BridgeSPA. 16

3.2 Basic interaction among bridge-related entities in a BridgeSPA setting. . . 17

4.1 A client using BridgeSPA to connect to a Tor bridge. 21

4.2 A failed authorization scenario with BridgeSPA. 22

4.3 The network configuration of the KnockProxy performance experiments (OpenWRT implementation). 25

4.4 Concurrent clients throughput measurements for the KnockProxy (OpenWRT implementation). 28

5.1 A BridgeSPA man-in-the-middle scenario. 31

5.2 BridgeSPA modified to prevent man-in-the-middle. 32

5.3 The network configuration of the DoorKeeper delay measurement experiment (standard implementation). 35

5.4 DoorKeeper delay measurement experimental configuration (standard implementation). 37

5.5 The network configuration of the DoorKeeper delay measurement experiment (OpenWRT implementation). 38

6.1 A visualization of 2500 initial TCP sequence numbers collected from a Windows XP SP3 machine. 40

6.2 A visualization of 2500 initial TCP sequence numbers collected from a Windows XP SP3 machine, separated by byte. 41

Chapter 1

Introduction

1.1 Tor and Bridges

Tor [9] is an open-source low-latency anonymity network that sees approximately 250,000 users per day [27]. Tor relies on volunteers to operate relays that forward end users' traffic. As shown in Figure 1.1, a client fetches a list of Tor relays from the Tor directory authority. The client selects three of these relays and establishes a layered, encrypted connection through them to reach his desired destination. Tor may be used to defeat some forms of Internet censorship. For example, a Canadian user wishing to access a web site that is blocked by Canadian Internet Service Providers (ISPs) can simply configure Tor to create circuits that must have exit nodes outside of Canada. Tor was initially designed, however, for anonymity and not censorship resistance. A list of current Tor relays is easily retrievable from centralized, publicly known directory authorities. This makes it trivial for an ISP to block all connections to Tor by blocking access to the IP addresses of all Tor relays. As of June 2011, China blocks Tor via this method [17]. Note that this is not the only way Tor has been blocked [8].

In an attempt to mitigate the ease of blocking Tor relays, unlisted relays (known as *bridges*) are available to provide alternate entry points to the Tor network. A bridge can be hosted on a specially deployed server, or it may be run on a home computer by a Tor user who has opted to help censored users reach Tor. All of the different components of the Tor network are available from the same binary executable, which is easily configured by the user. The standard Tor configuration tool makes it easy to operate as a bridge.

As shown in Figure 1.2, a bridge client operates in a similar manner as a regular Tor client. The bridge client must first connect through the bridge to access the list of Tor

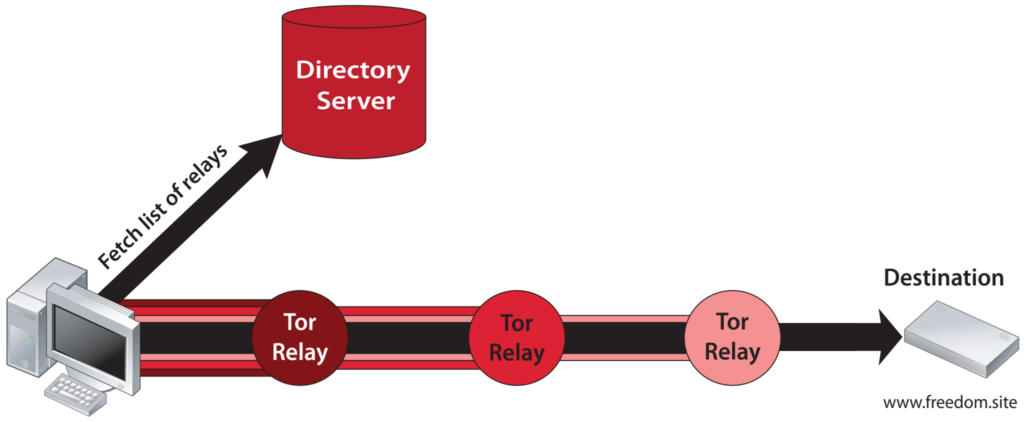


Figure 1.1: A client constructs a 3-node Tor circuit using the list of relays from the directory authority. The first node operates as a guard node, and remains persistent over many circuits. The second node is referred to as the middle node. The last relay is the exit node.

relays, and is then able to construct a 3-node circuit where the bridge is always the first node.

Bridges can be strictly unlisted (in which case information about the bridge is spread by word of mouth), or their descriptors can be distributed online by The Tor Project. A bridge descriptor always contains the bridge's IP address, and port. It can optionally include a hash of the bridge's public key (i.e., its fingerprint). As shown in Figure 1.3, the bridge authority keeps track of valid bridges, and the BridgeDB [19] provides the mechanisms for distributing bridge information through the web and by e-mail. This allows people in censored regimes to find bridge information through an encrypted webmail session. It is The Tor Project's aim to make it simple for users to find a few bridge descriptors, and at the same time to make it difficult for an adversary to find many bridges. To enforce this, The Tor Project restricts the distribution of bridge descriptors to three per week per 24-bit IP address prefix. Subsequent requests within a week will return the same bridge. However, the distribution mechanism does not account for attacks in which an adversary can gain control of many IP addresses through open proxies or botnets. In the case of a state-sponsored adversary, the adversary may have access to a significant number of IP addresses with distinct 24-bit prefixes.

It is clear that Tor bridges are not perfect. If one suspects that a particular host is operating as a Tor bridge, it is trivial to attempt to connect to the bridge to confirm this.

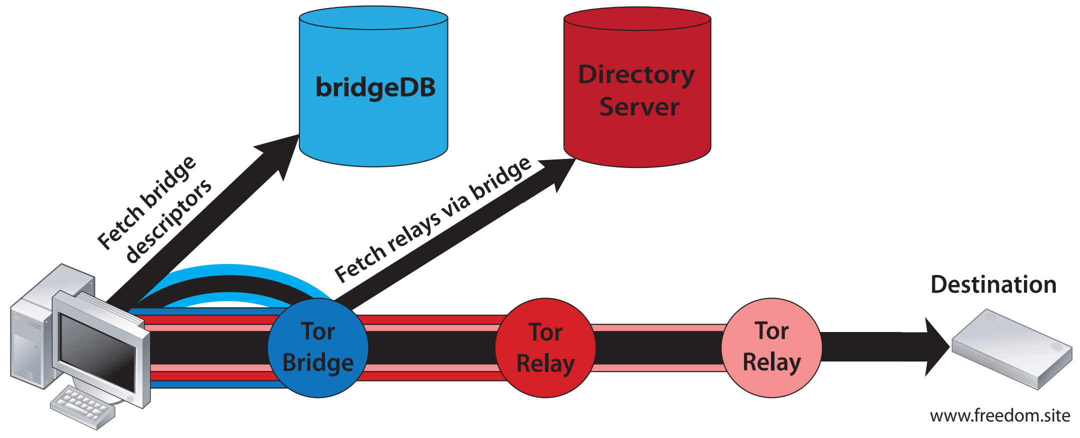


Figure 1.2: A bridge client constructs a 3-node Tor circuit. In this case, any connection to non-bridge Tor entities are tunneled through the bridge.

Bridges often run on predictable ports (such as 9001, 443), which means this confirmation can be performed with few resources. To gather a large list of bridges, a resourceful adversary can even test large blocks of IP addresses, or perhaps the entire IPv4 address space, on these ports. A less expensive way to gather bridges is to deploy a Tor relay. A Tor relay, unless operating strictly as a guard or exit node, can expect some of its clients to be Tor bridges. It is fairly simple to deploy a relay that is never selected as a guard or exit node by a standard Tor client.[1] A Tor relay operator can simply try to connect back to all clients using the method described above to determine which clients are bridges. An even more inexpensive method of determining bridges is to check if the client's IP address is listed as a Tor relay. If the deployed relay cannot be selected as a guard or exit relay, all of its clients must be guards or bridges. Thus if a client's IP address is not listed as a Tor relay it means the client must be a bridge. Note that this particular behaviour is likely to change in the future [6, 7, 21].

[1] A relay must explicitly be configured to become an exit node. Guard nodes are selected partially based on their reliability. To avoid becoming a guard, one can simply periodically restart the relay with a different relay identifier.

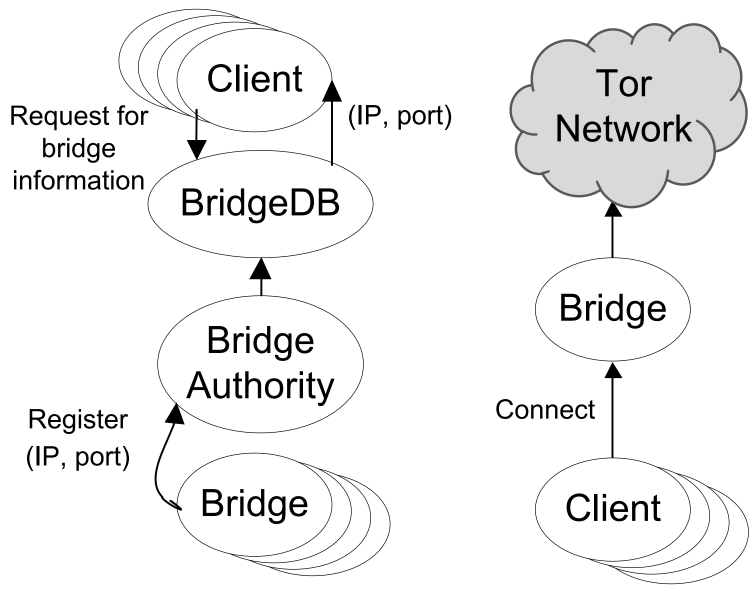


Figure 1.3: Basic interaction among Tor bridge entities. In the bottom left, a Tor bridge registers with the bridge authority and provides the IP address and port number for which it will accept bridge clients. In the top left, a client receives some bridge descriptors from the Tor BridgeDB. On the right, a client accesses the Tor network with the help of a bridge.

1.2 Bridge Aliveness Attack

McLachlan and Hopper [22] identified other issues with Tor bridges that could impact the anonymity of the bridge operator. Specifically, these issues apply to bridge operators who also use the bridge machine for web browsing. The attack they described is possible because it is easy to find a large number of Tor bridges, and a bridge always accepts connections from potential bridge clients while its operator is using Tor.

The attack considers a remote adversary who has a partial view of the Tor network. The adversary has enough resources to enumerate a large percentage of available Tor bridges over time. The adversary chooses a particular victim, who he knows is using Tor both as a client and to serve as a bridge. The goal of the adversary is to find the real IP address of this victim. The adversary must be able to detect some instances where the victim is online and using Tor, and he must have control over some content that the victim will access. It is not unreasonable to assume an adversary would have these abilities if he was, for example, the moderator of a message board that his victim is a member of.

The attack has three phases, which we collectively refer to as the "bridge aliveness attack." The **bridge discovery phase** involves collecting a large list of bridge addresses and descriptors for use later in the attack. Section 1.1 describes several ways an adversary could do this. When the adversary carries out the **winnowing phase**, he eliminates IP addresses from his list of bridges that he is sure do not belong to the victim. The attacker does this by querying all bridges in his list to see which are accepting connections ("alive") whenever he sees that his victim is online (e.g. making a post to a forum). This can be done by simply attempting to use each bridge as a client. Any bridge that is not accepting connections is not operated by the victim, since we are assuming the adversary knows the victim is online and using Tor. The adversary can continue the winnowing phase until he is left with just a few bridges. Finally, the adversary executes the **confirmation phase**, which uses a type of Tor circuit clogging attack to confirm the victim's IP address. To do this, the adversary embeds some content that he knows the victim will access. This content (e.g. an image) is hosted on a server, referred to as a burst server, which is controlled by the adversary. The rate at which the burst server tranfers data varies in a predictable way. For example, the server might intentionally restrict its transfer rate to 5 KiB/s for 15 seconds, and then transfer at an unrestricted rate for the following 15 seconds. While the victim is downloading from the burst server, the adversary connects to each remaining bridge. The adversary will attempt to correlate changes in the bridge connection's round-trip time (RTT) with the predictable rate changes followed by the burst server. If the adversary observes a strong correlation for a particular bridge, he can conclude that this bridge is likely to be the same client downloading from the burst server. The adversary now knows the IP address of a Tor client who believed she was anonymous.

The attack discourages Tor clients from opting in to serve as bridges since there is a chance their anonymity can be compromised.

McLachlan and Hopper [22] suggested a few methods to address different phases of this attack. To reduce the effectiveness of the winnowing phase, one idea they proposed was that a bridge could choose whether or not to serve clients based on a biased coin toss when the operator starts using Tor. This removes the close relationship between a bridge operator actively serving and using Tor as a client. Unfortunately, this results in fewer bridge resources available for clients.

One suggested method for mitigating problems in the bridge discovery phase is that a client should send a hash of the bridge's public key, discovered from the bridge authority, which must be verified by the bridge before the connection is accepted. This approach prevents a relay from easily attempting to connect to all of its clients to test whether or not they are serving as bridges. This change verifies that the client received a bridge descriptor from the bridge authority. The descriptor would still be valid indefinitely, however, since the hash does not change unless the bridge's public key changes, and hoarding bridges would still be possible.

Another suggestion to hinder bridge discovery is to require that the bridge's port is chosen at random. This makes bridge scanning more costly, but does not prevent scans from occurring.

1.3 BridgeSPA

This research focuses on BridgeSPA [32], a system that introduces an innocuous single packet authorization (SPA) system for Tor bridges. This makes it significantly more difficult to discover and hoard Tor bridge descriptors. It also eliminates the ability to arbitrarily query a bridge's aliveness. With these properties, BridgeSPA also mitigates the bridge aliveness attack.

At a high level, BridgeSPA introduces time-limited keys for bridges. As users request bridge information from the BridgeDB, BridgeSPA requires that the BridgeDB supplies this additional key. This key is in addition to the bridge IP, port, and fingerprint that the BridgeDB currently distributes. The key will only be valid for a time period defined by the associated bridge operator. An appropriate interval would be between 1 and 7 days. Clients using BridgeSPA must prove knowledge of this time-limited key in order to access a bridge or to even query aliveness of the bridge. This proof is presented in an innocuous single packet authorization (SPA) protocol [29,36], which allows the bridge to validate the key before responding. Failed authorization attempts from a client do not reveal aliveness, and a passive observer of the communication does not learn that the client is attempting to circumvent censorship.

The innocuous SPA protocol used in BridgeSPA is based on an existing system, SilentKnock [36]. SilentKnock, out-of-the-box, unfortunately does not work well with Tor bridges. For example, it requires that the server running SilentKnock maintains an explicit per-client counter that is synchronized with the clients. In general, an anonymizing service should not keep logs of its clients. Also, contrary to SilentKnock, all BridgeSPA clients for a specific bridge will use the same time-limited key. We have modified the SilentKnock protocol to allow it to work in a Tor bridge setting.

A client who wishes to access a bridge using the BridgeSPA protocol runs the BridgeSPA KnockProxy alongside the usual Tor client software. Similarly, bridges run the BridgeSPA DoorKeeper to authorize valid client connections. These processes run alongside Tor, and do not require changes to the Tor software.

In the next chapter, we look at some of the work BridgeSPA builds on. In Chapter 3 we describe the high-level BridgeSPA protocol. We explain and experiment on our implementations of this system in Chapter 4. In Chapter 5, we outline potential attacks on BridgeSPA, as well as countermeasures. We examine the Microsoft Windows implementation of TCP/IP for covert channels in Chapter 6. After this, we highlight potential future work in Chapter 7. Finally, we conclude in Chapter 8.

Chapter 2

Related Work

The core strategies of BridgeSPA build on existing work in port scan resistance and TCP/IP covert channels. An attacker usually launches a port scan to gather information about a target system, such as the operating system and specific services that are running. This helps an attacker determine what vulnerabilities may be present. One may limit a port scan's effectiveness by dropping all packets that do not arrive from a predefined whitelist of IP addresses. Maintaining a whitelist is not always desirable or even feasible, and other approaches are sometimes more appropriate.

In 2002, Barham et al. [4] proposed designs for a *silent authentication service* (SAS), which hides the existence of a service to a requester until they send specially crafted packets with an encoded secret key. They describe three possible designs for this. First, in Spread-Spectrum TCP a client sends a series of TCP SYN packets with the destination ports and initial sequence numbers (ISN) chosen such that the shared key is encoded. In the second design, Tailgate TCP, a client first sends a UDP packet containing an encoded, shared key. A TCP SYN packets follows immediately, which gives this design its name. Finally, in Option-Keyed TCP, a client sends a specially crafted TCP SYN packet with the shared key encoded in the ISN and TCP timestamp value. This single packet may be accepted by the server if the key was validated. While these designs do resist port scanning, the goal of this work was to make DoS attacks less effective since unwanted packets may be quickly dropped by the SAS before they are passed onto an application.

In 2003, Krzywinski [16] described *port knocking* as a simple mitigation of port scanning. A port knocking system will prevent access to a particular service until the requester sends a series of packets to a pre-defined sequence of ports. The port sequence is essentially a secret key for accessing a particular service. This is very similar to the Spread-Spectrum

TCP variant of SAS. Unlike SAS, port knocking was described with the intention for it to be easily implemented in Linux with simple shell scripts interacting with firewall rules. A client can similarly easily script a port knocking sequence with commonly installed tools like a telnet client. In 2006, Rash [29] described *single packet authorization* (SPA) as a variation of port knocking. An SPA system conceals the existence of a particular service until the requester sends a UDP packet to a particular host with an appropriate payload. This is very similar to the idea of Tailgate TCP. All of these systems successfully resist a port scan; however, an adversary who is capable of passively monitoring communications could infer the existence of an SAS, port knocking or SPA service in these scenarios by recognising communication patterns that are not characteristic of normal TCP connections.

A number of researchers have studied the TCP/IP suite for opportunities to implement covert channels [11, 23, 30]. In 1997, Rowland [30] described how the IP ID field in an IP packet as well as the ISN in a TCP packet can be used to encode information. This work was later strengthened by Giffin et al. [11] to use the TCP timestamp field, and also to use message authentication codes to encode the covert messages. This is similar to what Barham et al. did with SAS, but without the intention of being covert to a passive observer. Murdoch and Lewis [23] later presented an analysis of how different versions of Linux and OpenBSD select ISN and IP ID values. Since parts of these values are not uniformly random, an appropriate distribution must be considered by covert messaging systems to avoid leaking information. Related to TCP/IP covert channels, Goh et al. [12] describe an implementation of protocol-based key recovery. They show how user-chosen random fields in protocols such as TLS and SSH can be chosen by one of the parties such that a passive adversary can discover the secret key protecting the session.

In 2007, Vasserman et al. presented SilentKnock [36], a form of SPA and SAS that takes into consideration the aforementioned work on TCP/IP covert channels. They also presented a formal model against which one may evaluate innocuous SPA systems. A client attempting to access a service guarded by the SilentKnock daemon (`sknockd`) must initiate a specially constructed TCP SYN packet to the target IP address and port. This is similar to Option-Keyed TCP described above, but with the TCP header fields chosen in a way such that using SilentKnock is provably undetectable to a passive observer. This packet contains a calculated MAC encoded in the lower 3 bytes of the ISN value and the lower byte of the TCP timestamp field. In its simplest version, the MAC is keyed with a pre-established long term key and is applied to a per-client counter value, as well as source and destination IP address and port pairs. The per-client counter needs to stay synchronized between SilentKnock clients and `sknockd`. Upon receiving a TCP SYN packet, `sknockd` is able to recompute the expected MAC value using its own copy of the pre-established long term key and per-client counter value.

SilentKnock chooses to use the lower 3 bytes of the ISN value and the lower byte of the TCP timestamp field since these values are uniformly random in Linux 2.6. The highest-order byte of the ISN in Linux does not have the same property. While the TCP timestamp does not need to be related to the current system time, in Linux 2.6 the timestamp values are consistent across connections. In the case where the last byte of a TCP timestamp needs to be increased to correctly encode the MAC output, it should not be the case that an immediate subsequent connection contains a lower TCP timestamp value. To avoid this, SilentKnock clients will also delay themselves accordingly before sending a TCP SYN packet when its TCP timestamp has been increased.

As mentioned above, BridgeSPA uses a form of SPA and SAS based on SilentKnock. BridgeSPA also attempts to use the lower 3 bytes of the ISN value, and the lower byte of the TCP timestamp field. When the TCP timestamp field is not present, BridgeSPA will use all 4 bytes of the ISN value. We discuss our reasoning for this in Chapter 6. In general we will assume that the TCP timestamp field is present. We also assume that no intermediary router will remove the TCP timestamp. Unlike SilentKnock, BridgeSPA does not try to maintain a counter for each client to prevent replay, but instead includes the loosely rounded UTC time in the MAC pre-image. As a trade-off, by including the time instead of a synchronized counter we introduce a small opportunity for replay. Bridge clients should not possess unique keys, and it is not appropriate to require Tor bridges to maintain an access counter for each client IP address. We describe our implementation with more detail in Chapter 4. In Chapter 2.1 we expand this discussion by comparing BridgeSPA to other popular port knocking implementations.

For convenience, the structure of a TCP/IP packet header highlighting fields that may be used for covert channels is shown in Tables 2.1 and 2.2.

Specific to Tor, there have been investigations [13, 26] into including keys with bridge descriptors for the purpose of making bridges more difficult to detect. Both of these works propose that a client should send this secret to a bridge after a standard TLS connection has been established. If the bridge does not receive the secret, or the secret is invalid, the bridge will act like a regular web server. Note that this alone does not protect against the bridge aliveness checks needed for the bridge aliveness attack. We describe how this method will, however, protect against an active adversary hijacking a client's bridge connection in section 5.2.

There has been a proposal to use port knocking and SPA for Tor bridges to make them more resilient against detection [1]. This suggestion includes using DNS packets as a transport method for SPA. While this idea would be much simpler to implement, we feel it is necessary for the traffic that a bridge client generates while using Tor to be completely

Table 2.1: An IP packet header. We use *italics* for values that can be used for covert channels, as described by Murdoch et al. [23].

0								1								2								3							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Version				IHL				Type of Service								Total Length															
Identification (IP ID)																*Flags*			*Fragment Offset*												
Time to Live								Protocol								Header Checksum															
Source Address																															
Destination Address																															
Options																								Padding							

Table 2.2: A TCP packet header. We use *italics* for values that can be used for covert channels, as described by Murdoch et al. [23]. Values that are in **bold** are used by BridgeSPA.

0								1								2								3							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Source Port																Destination Port															
Sequence Number																															
Acknowledgement Number																															
Data Offset				Reserved				TCP Flags [CUAPRSF]								Window Size															
Checksum																Urgent pointer															
*Options and **Timestamp***																								Padding							

innocuous. A DNS request to a particular IP always being followed by a TLS connection to that same IP may stand out to an adversary who suspects that a client is using Tor bridges.

2.1 Comparison with Other Knocking Systems

In this section we compare and evaluate BridgeSPA against popular knocking systems in terms of properties, protocol, and implementation. Some port knocking systems have similarities to the system implemented by BridgeSPA.

2.1.1 Knockknock

Knockknock [20] is an SPA system created by security researcher Moxie Marlinspike. Its primary goal was to eliminate the trivial replay attacks that could occur in earlier port knocking systems if an adversary passively monitored a successful authorization. Marlinspike imposed several more restrictions on his design, such as:

- The implementation should be written in a memory-safe language.
- The implementation should not run in the kernel
- The implementation should not inspect every packet

Knockknock's protocol consists of a client sending a TCP SYN packet to a particular port, which is monitored by a the `knocknock-daemon`. These TCP SYN packets contain a MAC payload that the daemon checks before opening another port to accept connections. Knockknock implements this with a firewall rule that logs TCP SYN packets to a file. An unprivileged thread looks in this file for a specific MAC output encoded in the packet's headers.

While this is a very good design, it does not have the same goals as BridgeSPA. Knockknock is similar to BridgeSPA in that they both use packet header fields and MACs to encode values, but that is the extent of the similarity.

When a client sends a valid SPA packet, the `knocknock-daemon` simply allows the desired port to be accessible by the client. Another TCP connection must be initiated after the client's knocker has executed. To a passive observer, this protocol is not covert.

It may be possible to use raw sockets to respond to and initialize the client connection after reading a valid logged TCP SYN. This would require superuser privileges. Also note that the knockknock-daemon in this case would still likely incur a significant delay, compared to that of the BridgeSPA DoorKeeper.

Knockknock targets the GNU/Linux operating system, and is implemented in Python.

2.1.2 Fwknop

Fwknop [28] is a popular, practical, and well-supported SPA system. It is compatible with Linux, Mac OS X, BSD, and has partial support under Windows (with Cygwin). It supports hiding services behind NAT devices, and allows flexible restrictions on incoming SPA packets.

Fwknop uses `libpcap` to passively listen for packets, but also supports modes similar to Knockknock. That is, it will accept a specifically formatted iptables or tcpdump output stream.

The Fwknop protocol consists of a client sending a value that proves the possession of a key, encoded somewhere in the body of an IP packet. By default, it operates using AES for symmetric encryption and uses UDP as the transport layer.

Alternatively, Fwknop may be configured to use asymmetric cryptography. For example, a client sends a signed GPG message that is encrypted for a particular GPG key-holder. This would be the payload of the knocking packet. Fwknop can also establish a TCP connection to send its payload, or use an ICMP packet.

The Fwknop is implemented in C, and is available in many operating system package repositories, including OpenWRT.

Fwknop also does not intend to be covert. By default, the client sends a UDP payload to a particular port. To a passive observer this would be trivial to observe. We also see that Fwknop's goals and default behaviour are quite different from Knockknock. Fwknop intends to be flexible, and most qualities of its protocol are configurable. It would be difficult to introduce covert SPA capabilities to Fwknop without breaking compatibility with some operating systems.

Chapter 3

The BridgeSPA Protocol

BridgeSPA has two major goals. As suggested above, our first goal is to mitigate the ability to easily detect bridges and carry out the bridge aliveness attack. Specifically, we wish to remove the ability for a client to easily identify a particular host as a bridge, connect to the bridge, or even determine if the bridge is online unless he has recently obtained a key. This significantly increases the amount of network resources an attacker must have to carry out the bridge aliveness attack, since bridge information received from The Tor Project will expire.

Our other major goal states that our changes and additions should not provide additional information to an adversary who is attempting to detect clients who use bridges. Currently Tor traffic is distinguishable from non-Tor traffic and the Tor community is working to address this [3]. Our work should not hinder their efforts in any way. A man-in-the-middle observing a client connecting to a bridge should not learn that a Tor connection is being established simply because of the changes we propose.

Some secondary goals are as follows:

- **Minimal communication overhead:** Bandwidth is frequently a limiting resource for Tor. It is therefore important to minimize the communication overhead our protocol imposes on bridge operators, clients and authorities.

- **Preserve the "unlisted bridge" mode:** Currently, bridges can operate without having their descriptors listed at a bridge authority. This provides the ability for a bridge operator to manually distribute her bridge information to specific users of her choice. This is a useful ability that we must preserve.

- **Maximize bridge uptime:** To best utilize bridge resources, we also want to allow a bridge to serve whenever it is available to do so.
- **Minimize assumptions about Tor protocols:** Tor is an actively evolving network. Our protocol should be agnostic to the specifics of other Tor protocols.

3.1 Adversarial Model

As our two primary goals are addressing different types of threats, the adversarial models we must consider are also different. While addressing the bridge identification and the bridge aliveness attack, we consider a remote non-global adversary who is capable of querying for a reasonably large number of bridge descriptors from bridge authorities over time. This adversary may perform aliveness tests by attempting to connect to bridges as a bridge client. This adversary would also have access to timestamps of contributions and control some content on a website where the bridge operator makes pseudonymous contributions, in order to carry out the bridge aliveness attack.

When considering whether a bridge client may be identified as a Tor user, our adversary is much more capable. We assume an active adversary with full control of the local network in which the client is present. She is capable of monitoring, injecting, replaying, shaping and dropping packets but only within her network bounds. This adversary has no view or control of the outside network, where Tor relays and bridges operate.

3.2 Protocol Details

In this section we describe the life cycle and high-level interaction of bridges and clients using the BridgeSPA protocol. BridgeSPA is an authorization protocol that allows a client to connect to bridges for which they have a valid descriptor but prevents an adversary from hoarding bridge descriptors over long periods of time. BridgeSPA accomplishes this while also not permitting bridge aliveness checks from adversaries who do not possess valid descriptors.

The BridgeSPA protocol utilizes a pre-shared key to generate MACs that are used to determine the legitimacy of a bridge connection request. That is, they ensure that the person making the request has recently obtained a valid key. We consider three main phases of a bridge's life cycle that are affected by BridgeSPA: bridge registration, bridge

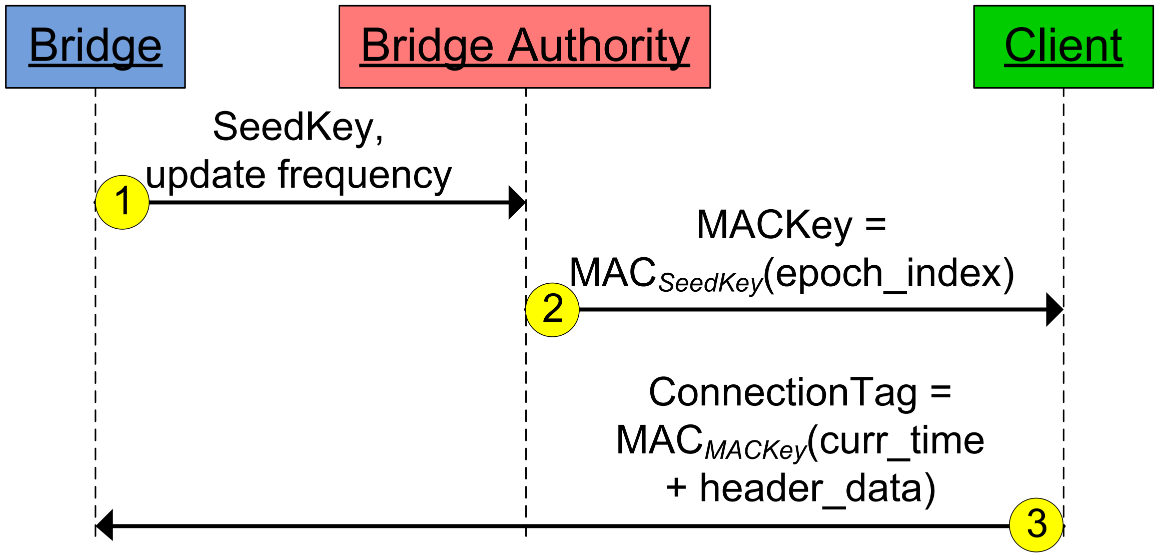

Bridge
Bridge Authority
Client
SeedKey, update frequency
1
MACKey = MAC_SeedKey(epoch_index)
2
ConnectionTag = MAC_MACKey(curr_time + header_data)
3

Figure 3.1: Changes to Tor bridge-related communication in BridgeSPA. 1. Upon registration with the bridge authority, a bridge includes a SeedKey and update frequency. 2. The bridge authority distributes bridge information along with the current MACKey, derived from the SeedKey. This information typically passes through the BridgeDB (not shown). 3. A client uses the MACKey to create a ConnectionTag, which must be included by the client when connecting to this bridge.

descriptor requests and client connection. Figure 3.1 shows a high-level outline of the changes made to the bridge's life cycle to support BridgeSPA.

Currently during bridge registration, a bridge that wishes to be publicly listed communicates with the bridge authority directly and provides a descriptor with information needed for connecting to it. This includes the IP address, port, and optionally, a fingerprint. In BridgeSPA, the SeedKey (a 256-bit random value) and an associated update frequency are also included. This change is highlighted in Figure 3.2. A reasonable value for an update frequency would be between 1–7 days, and may be configured by the bridge operator.

The bridge and bridge authority are each able to calculate the current epoch index from the update frequency and the current time. An epoch index is defined as the current Unix time divided by the update frequency. Using the SeedKey and the current epoch index, a bridge and the bridge authority may independently compute short-lived MACKeys valid for any particular time period. A MACKey is what a client, using the KnockProxy, uses

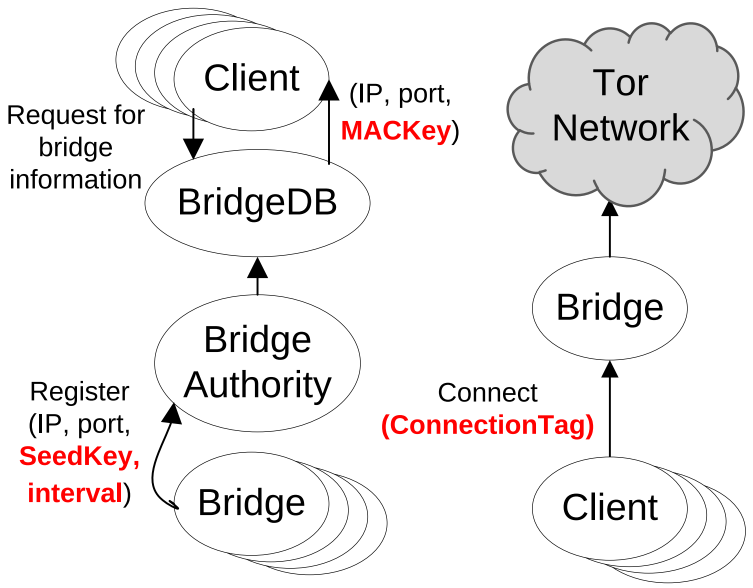


Figure 3.2: Basic interaction among bridge-related entities in a BridgeSPA setting. Changes are highlighted in bold text.

when he connects to a bridge to demonstrate that he has recently obtained the bridge descriptor from an appropriate source.

To initiate a connection, the client first generates a ConnectionTag that is embedded in the first network packet (the TCP SYN) that is sent to the bridge. The ConnectionTag is a MAC of the current time and header data from the SYN packet, keyed with the MACKey. The time used is represented as UTC, rounded down to the minute. This is done to make replay attacks more difficult for an active adversary. The header data includes the source and destination IP address and port pairs, and the IP Identification field. The ConnectionTag is used instead of sending the MACKey directly to avoid replay attacks.

A bridge, using the DoorKeeper, monitors incoming TCP connection requests. When the DoorKeeper identifies an incoming bridge connection request it is able to check the embedded ConnectionTag using its own copies of the MACKey, current time, and header data. To gracefully deal with edge cases, the DoorKeeper also calculates and compares ConnectionTags of the previous and next minutes before dropping a packet. This means there are three ConnectionTags that the DoorKeeper will accept at any given time.

The client embeds a ConnectionTag using headers in the the TCP SYN packet as a covert channel. As mentioned above, following a strategy based on SilentKnock [36], to a passive observer the request is indistinguishable from a connection request not containing authorization information.

Chapter 4

BridgeSPA Implementations

We have developed two proof-of-concept implementations of BridgeSPA to test our protocol and the feasibility of deployment. We have implemented a version intended to run in a standard GNU/Linux desktop or server environment, which we refer to as the "standard" or desktop implementation.[1] We have also developed a version that targets an OpenWRT GNU/Linux environment. As mentioned, our implementations require no changes to the Tor client or bridge software. To support a deployed instance of BridgeSPA, however, the Tor bridge authority and BridgeDB would need to be modified as shown in Figure 3.2.

Our design targets Linux, with kernel version 2.6.4 or later due to our reliance on the `libnetfilter_queue` library [34]. This library provides a user-space API for manipulating network packets in flight. Our implementations of both the DoorKeeper and KnockProxy use `libnetfilter_queue` to implement their respective parts of the BridgeSPA protocol. On the client side we also developed a small tool to assist with configuration. This tool will take a BridgeSPA bridge descriptor, optionally configure Tor to use the bridge, and pass along the information to the running KnockProxy process. The KnockProxy allows this by accepting connections on a configured port.

When the KnockProxy receives bridge descriptors, it keeps the MACKey for later use and adds an iptables rule to ensure TCP packets intended for this destination will be available through the `libnetfilter_queue` APIs. When the KnockProxy encounters a TCP SYN packet intended for a Tor bridge, it makes the necessary changes to the ISN and TCP timestamp based on the calculated ConnectionTag. The ConnectionTag is only calculated once per connection. When it is calculated, an offset from the original ISN

[1]The standard implementation is available at `http://crysp.uwaterloo.ca/software/` under a free licence.

as well as the original TCP timestamp value gets stored for use later. The KnockProxy associates these values with the TCP connection, so they may be easily looked up as needed. All subsequent packets sent to and received from the bridge also get modified to have their sequence and acknowledgement numbers adjusted appropriately, by applying the offset value. Only the first TCP timestamp value sent by the client and the first TCP echo value sent by the server need to be adjusted by the KnockProxy. This is because the KnockProxy will actually delay itself until the time is consistent with lower byte of the timestamp value, which is similar to the behaviour of SilentKnock. Whenever the KnockProxy modifies the contents of the TCP header it must also recalculate the TCP checksum.

The ConnectionTag is a SHA256-HMAC output, truncated to 32 bits. This MAC is keyed with the MACKey from the bridge descriptor, and applied to data found in the TCP and IP headers, along with a rounded value of the current time in UTC. It requires, however, that the client and bridge are both loosely synchronized with an accurate NTP server. As previously mentioned, the current time is rounded to the nearest minute.

The DoorKeeper receives the SeedKey and time interval as command line parameters. The DoorKeeper similarly instructs iptables to queue SYN packets arriving at the pre-specified bridge port. As SYN packets arrive, the ISN and timestamp are checked to determine if they contain a valid ConnectionTag. To calculate a ConnectionTag, the DoorKeeper also needs to find the current MACKey from the time interval and SeedKey. To avoid having to do this while processing a packet, a separate thread will periodically update this value. Similarly, another thread updates the current time rounded to the nearest minute, to avoid an unnecessary system call while processing packets. The packet is allowed to continue if the ConnectionTag matches. If the match fails, the packet is dropped and will not be processed by the operating system's TCP stack.

If the DoorKeeper or KnockProxy receive an interrupt or termination signal, they will attempt to remove the configured firewall rules.

The DoorKeeper implementation meets the goal of preventing aliveness checks, since packets that are rejected will not return any sort of response. This is outlined in Figure 4.2. When a connection is closed, the client associated with that connection can no longer communicate with the bridge without initiating another connection with a valid ConnectionTag.

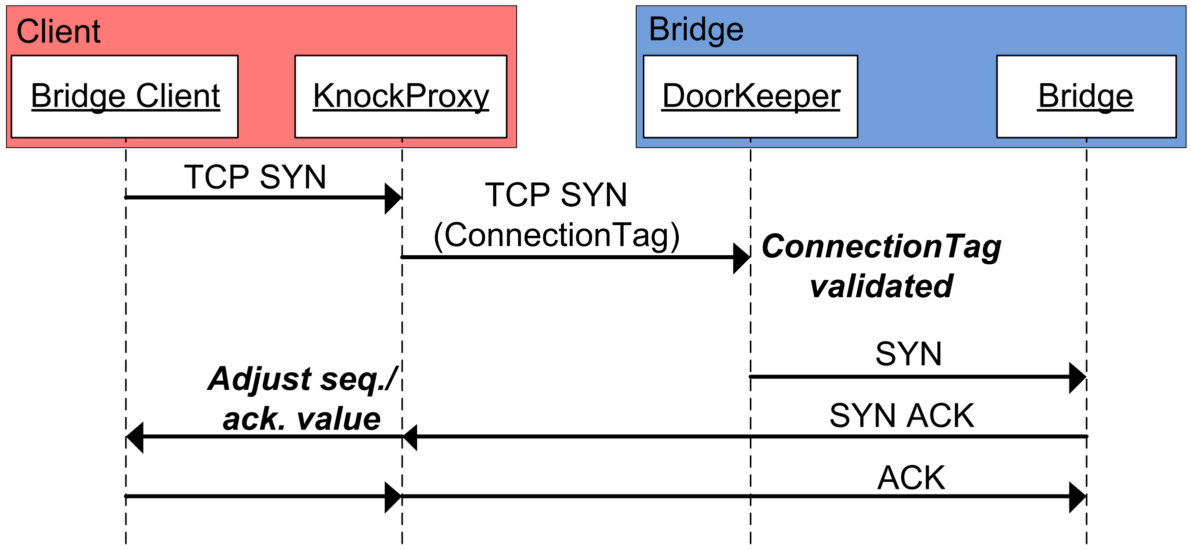


Figure 4.1: A client using BridgeSPA to connect to a Tor bridge.

4.1 Unlisted Bridges

In Chapter 3, we described how a client who received bridge information from The Tor Project may connect to a bridge using the BridgeSPA protocol. If an operator runs an unlisted bridge, she must manually send some information to her clients. We outline three possibilities for this type of scenario, two of which are already compatible with our proof-of-concepts.

First, the bridge operator can share a SeedKey and update frequency with the client directly using an out-of-band channel. The client simply generates the current MACKey, and then configures the KnockProxy following the same methods described above. Second, the bridge operator could simply share the current MACKey, the same way The Tor Project would distribute bridge information with the BridgeSPA protocol. Both of these are possible with our proof-of-concept implementations. The latter requires that the operator continues to send updated MACKeys when the specified epoch expires. Third, the bridge operator could offer a SeedKey or MACKey that is unique to each client's IP address. This is not currently implemented, and it may be less convenient for clients who do not have static IP addresses. Note that a future DoorKeeper design implementing this scenario ideally should not have to store a list of client IPs and their respective keys; including the

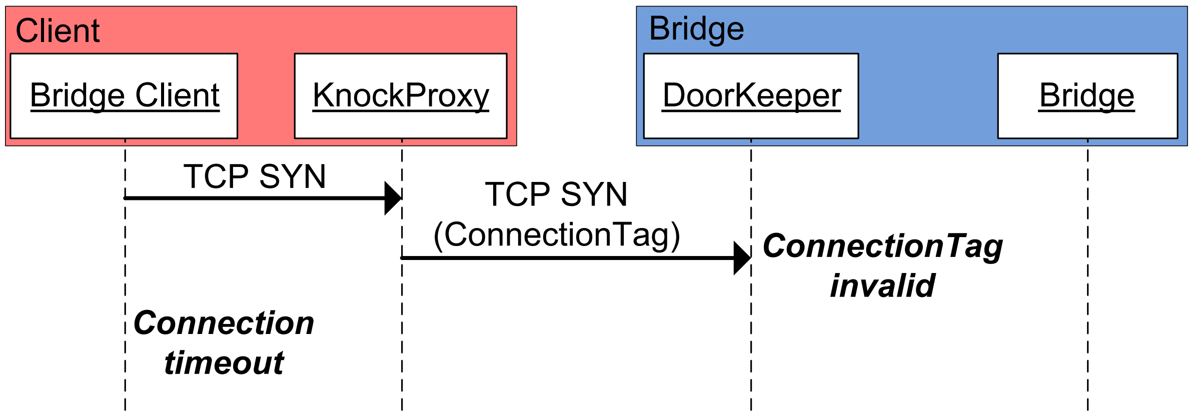


Figure 4.2: A failed authorization scenario with BridgeSPA.

client IP in the computation of the MACKey, for example, would suffice.

4.2 GNU/Linux Standard Implementation

4.2.1 Performance Impact

When a client uses the KnockProxy, all packets are modified as they are sent and received by the client. This could impact performance, especially for high-throughput connections. We executed an experiment to measure the performance impact of KnockProxy. We downloaded a 6.3 MiB file 8 times with and without the KnockProxy, over a 100 Mb/s LAN and from a remote server on the Internet. While our motivating use case for BridgeSPA does not necessarily make sense in a LAN, we include a LAN experiment to capture a high-throughput scenario with a guarantee of no interference from other traffic.

We see in Table 4.1 that the performance impact introduced by the standard KnockProxy implementation is negligible. In the LAN scenario, the traffic rate only degrades by 1.7%. In the Internet scenario, there was no statistically significant difference in the rates. CPU and RAM usage during this experiment were also negligible. While downloading, KnockProxy's share of CPU usage never exceeded 10%, and the RAM usage stayed consistent at 20 MiB.

We see that the standard implementation of the KnockProxy is unlikely to introduce a performance impact for users. Internet speeds (and Tor speeds) would need to dramatically

Table 4.1: We downloaded a 6.3 MiB file 8 times with and without the KnockProxy, over a 100 Mb/s LAN and from a remote server on the Internet. The test client was a Thinkpad X201 with a Core i5 540M 2.53 GHz CPU and 8 GiB RAM, running Ubuntu 11.04 in VirtualBox 4.0. The client's host operating system was Windows 7 Professional.

		Download Rate ± stddev	Upload Rate ± stddev
LAN	Without KnockProxy	8090 ± 50 KiB/s	5.34 ± 0.04 KiB/s
	With KnockProxy	7950 ± 40 KiB/s	5.25 ± 0.03 KiB/s
Internet	Without KnockProxy	583 ± 2 KiB/s	0.374 ± 0.004 KiB/s
	With KnockProxy	585 ± 3 KiB/s	0.377 ± 0.002 KiB/s

increase in order for KnockProxy to become a bottleneck.

4.3 OpenWRT Implementation

Our initial GNU/Linux implementation has three apparent usability flaws. First, not all bridge users run GNU/Linux. Ideally a user should be able to use the KnockProxy no matter which operating system he runs. Second, the implementation does not work correctly when running behind a home network address translation (NAT) device. Almost all households that have multiple Internet-connected devices have deployed some type of NAT device. If a user is behind a NAT, the source port and IP address that the KnockProxy observes in the outgoing TCP SYN packet to calculate the ConnectionTag will not necessarily be the same source port and IP address that the DoorKeeper sees. The DoorKeeper would calculate a different value for the expected ConnectionTag. Finally, both the KnockProxy and DoorKeeper need to run with superuser privileges. This is not ideal, since if there is a vulnerability in KnockProxy or DoorKeeper an attacker could gain control of the user's entire system.

OpenWRT is a project that offers an alternative firmware for home NAT devices. This firmware is a lightweight GNU/Linux distribution. We have created a version of the KnockProxy and DoorKeeper specifically for OpenWRT. With this version, clients running any operating system may use the KnockProxy in a NAT scenario. Both the KnockProxy and DoorKeeper would still need to run as root on the router. If an attacker exploited a vulnerability in either of these programs he could still take control of the router. Assuming other systems connected to the router are secure, an attacker's access to user data would be limited.

4.3.1 Implementation Details

The KnockProxy implementation for OpenWRT consists of the same source code as the standard GNU/Linux implementation. The work to produce the OpenWRT implementation mostly involved configuring OpenWRT to support BridgeSPA, rather than modifying the BridgeSPA software. OpenWRT simply requires that we follow their standard for integrating with their build configuration system. This also ensures that dependencies will be identified and handled. However, using `libnetfilter_queue` for packets that are being handled by a NAT makes it difficult to capture packets with the same source and destination IP addresses and ports that the remote server will observe. This creates a challenge when determining which values should be used while calculating a ConnectionTag, as well as when pairing incoming packets and outgoing packets that are a part of the same TCP session.

Another issue is that, since the intention is to support a wide variety of clients, outgoing packets may not have the TCP timestamp option present for modification. It would be possible to add a TCP timestamp to outgoing packets, but this would require a full implementation of the TCP timestamp (RFC 1323). As an alternative, we have integrated a proxy to force the router to establish new connections. From the client's point of view, his connections appear to work the same way as under a NAT. However, unlike with NAT, since all external connections are actually originating directly from the router, they can be guaranteed to carry a TCP timestamp. Further, when intercepting these packets, the KnockProxy will observe the same source and destination IP addresses and ports that the remote party will see.

We use a firewall rule to redirect client connections to a custom transparent proxy application. The method we use is distinct from the GNU/Linux implementations of "tproxy", which intends to supply means to intercept and proxy connections while avoiding modifications to the source and destination portions of TCP and IP packets. That is, if a client's IP address is routable by some remote server, "tproxy" will not insert the router's own IP address in the client's outgoing packets. Conversely, with our system the intention is to always replace the client's IP address with the router's.

A user executes and configures the transparent proxy program on the router as a separate process from the KnockProxy. The same client configuration tool as with the GNU/Linux desktop implementation may be used to register bridges with KnockProxy.

Support for our test router (Buffalo WZR-HP-G300NH2) was added only recently as a patch to the unreleased OpenWRT branch (i.e., SVN trunk). Specifically, we tested our implementation with the OpenWRT trunk revision 29283. While we could not deploy an

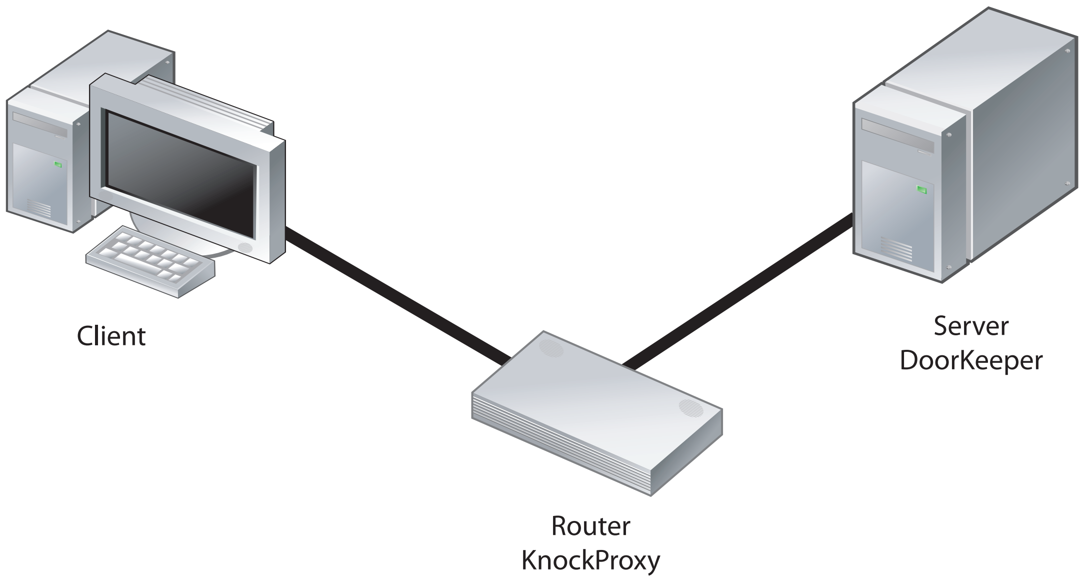


Figure 4.3: The network configuration of the KnockProxy performance experiments (OpenWRT implementation).

officially released version of OpenWRT to our router, we have tested our implementation in a virtualized environment with the latest stable branch of OpenWRT, version 10.03 ("Backfire") SVN revision 28637. We expect our implementation to be compatible with other routers that have similar specifications to our test router, as listed below.

4.3.2 Performance Impact

We look at the performance impact of the transparent proxy and KnockProxy running in OpenWRT, separately and together.

The experimental configuration for all experiments in the following subsections is shown in Figure 4.3. The client is a Thinkpad X201 with a Core i5 540M 2.53 GHz CPU and 8 GiB RAM, running Microsoft Windows 7 Professional. The server is a desktop with an AMD Phenom II 965 3.4 GHz CPU and 4 GiB RAM, running GNU/Linux 2.6.32. Our test router has an Atheros AR7242 400 MHz CPU and 64 MiB RAM. The speed of the wired ethernet connections is 100 Mb/s.

Table 4.2: Clients download a 6.3 MiB with `scp` 8 times under each scenario. The scenarios vary whether the transparent proxy and KnockProxy are running.

	Download Rate ± stddev	Upload Rate ± stddev
No BridgeSPA processes	8100 ± 200 KiB/s	5.0 ± 0.1 KiB/s
Transparent Proxy (TP)	6040 ± 90 KiB/s	3.73 ± 0.06 KiB/s
TP and KnockProxy	2700 ± 40 KiB/s	1.67 ± 0.03 KiB/s

Table 4.3: Clients download a 6.3 MiB with `wget` 8 times under each scenario. The scenarios vary by network capacity and whether the transparent proxy and KnockProxy are running.

Server Speed Limit	Configuration	Download Rate ± stddev
1536 KiB/s	No BridgeSPA processes	1500 ± 200 KiB/s
1536 KiB/s	Transparent Proxy (TP)	1500 ± 300 KiB/s
1536 KiB/s	TP and KnockProxy	1400 ± 200 KiB/s
600 KiB/s	No BridgeSPA processes	590 ± 30 KiB/s
600 KiB/s	TP	590 ± 30 KiB/s
600 KiB/s	TP and KnockProxy	590 ± 40 KiB/s

Single Connection Throughput

We observe that both the transparent proxy and KnockProxy introduce a significant bottleneck while operating on a 100 Mb/s network, as shown in Table 4.2. With the transparent proxy running, performance suffers by approximately 25%. When we add the KnockProxy, an additional 55% degradation occurs.

To measure how the performance degradation is related to the network speed capacity, we introduce a global speed limit on the server. For these experiments, clients downloaded files from an instance of the Lighttpd web server [15], with the `server.kbytes-per-second` configuration setting changed as desired. Note that this value introduces a per-server speed limit, and not a per-connection speed limit.

Table 4.3 shows that performance degradation occurs less dramatically when the speed capacity of the connection is lower. We included the 1536 KiB/s scenario since it is the approximate average household Internet connection speed in Canada and the United States [24]. We included 600 KiB/s to represent a generous upper bound of Tor stream speeds [25]. With a 1536 KiB/s connection, clients only suffer a 7% performance loss compared to when not running the KnockProxy and transparent proxy. This is significantly better than the 66% speed degradation we observed with a 100 Mb/s line connection. In the 600 KiB/s scenarios, the average speeds are not affected by the transparent proxy and the KnockProxy.

Multiple Connections Throughput

A home router is capable of supporting many concurrent devices and connections. As we know our OpenWRT test device's resources are limited, we measured to see if concurrent clients would further impact the network performance while running the transparent proxy and KnockProxy. We tested 2, 4, and 8 clients downloading the same 6.3 MiB file from a Lighttpd server. We first tested with no speed limit configured aside from the 100 Mb/s line speed, and then with a global 1536 KiB/s limit in Lighttpd. Clients performed three trials in each scenario, and an average is plotted on Figure 4.4.

Figure 4.4 shows a clear degradation in both types of connection speeds when increasing from a single client to multiple clients. In the 100 Mb/s scenario, moving from two to four clients only introduces another 5% slowdown, and the 1536 KiB/s connections show almost no change. Similarly, increasing from four to eight clients shows little difference in the transfer rate.

Improving Performance

We have not yet taken any steps to optimize the KnockProxy specifically for devices with limited resources. One optimization could be to move the KnockProxy into kernel space. Since a user in an OpenWRT environment typically reinstalls their router's entire filesystem image for updates, it would not be unreasonable to require additions to the kernel in the image. SilentKnock [36] takes this approach to decrease the delay added by `sknockd`.

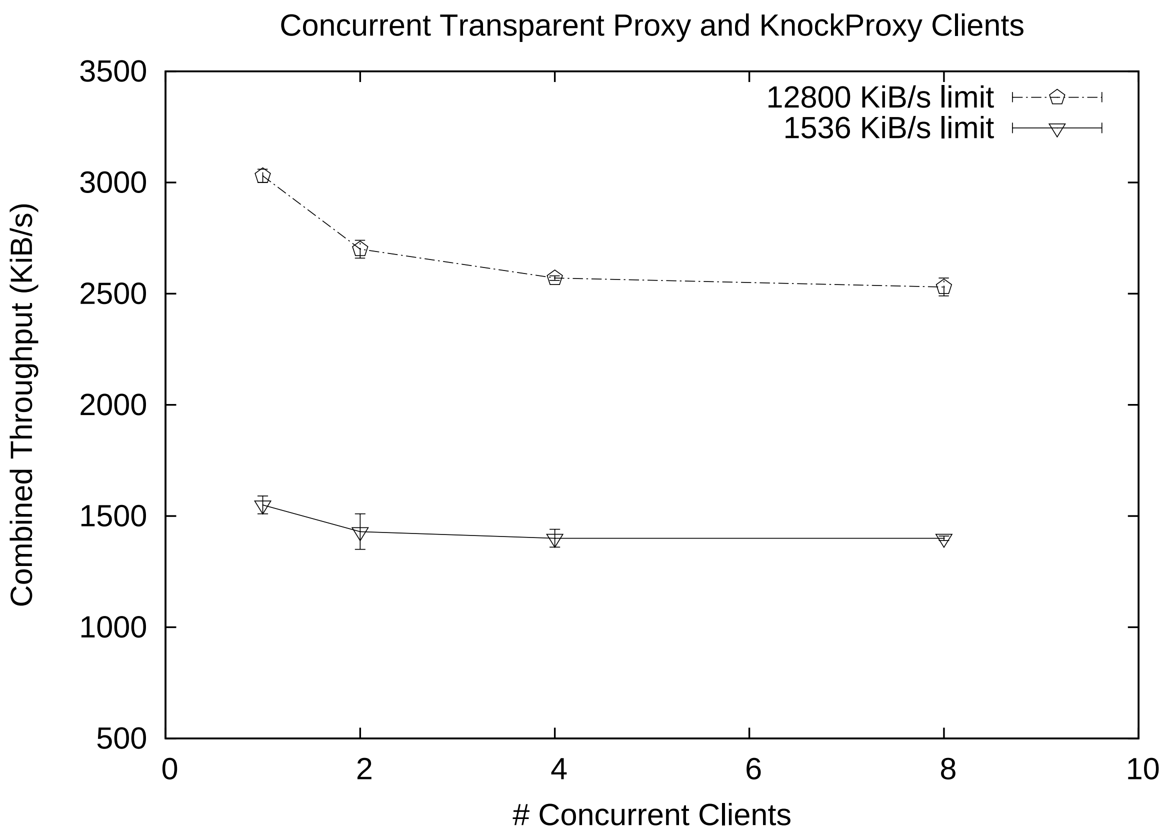


Figure 4.4: Concurrent clients downloading with `wget` through a router configured with KnockProxy and the transparent proxy. All clients were executed on a Thinkpad X201 with a Core i5 540M CPU and 8 GiB RAM, running Microsoft Windows 7 Professional. The depicted error bars represent one standard deviation above and below the averages. Note that the first data point in the 12800 KiB/s line is different from the equivalent data point in Table 4.2 due to the different transfer protocols.

Chapter 5

Attacks

In this chapter, we analyze BridgeSPA's effectiveness by discussing some attacks that are possible under the adversarial models defined in section 3.1.

5.1 Bridge Aliveness and Enumeration Attacks

We can attempt to re-apply the bridge aliveness attack from section 1.2. In the discovery phase, an attacker was able to rely on the fact that bridge information could be collected over a large timespan and remain mostly valid. This is no longer the case, since the MACKey for each bridge will change between the epochs set by the update frequency. As a result, for an attacker to obtain enough bridges in a single time interval for the winnowing phase to be effective, she must have access to a large number of IP addresses *in a short time*. Furthermore, due to constraints in the bridge information distribution protocol, these IP addresses must have distinct /24 network addresses. Thus, the bridge aliveness attack is still possible but requires significantly more resources. This limits the attack to an adversary who has access to a large number of IP addresses, which we consider to be a substantial improvement.

A determined adversary who thinks that a particular bridge is concealed by BridgeSPA could try to determine aliveness by guessing the correct ConnectionTag. As mentioned in Chapter 3, a DoorKeeper will accept three ConnectionTags at any time. As a ConnectionTag is 32 bits long, this leaves an attacker with an expected 2^{30} guesses before finding a valid ConnectionTag. It would be possible to blacklist an IP after many incorrect guesses, but this may be better suited as a firewall rule as opposed to an extension to BridgeSPA.

Also note that if an adversary is flooding a particular host with ConnectionTags, this would appear to the host as a SYN flood, something that is often mitigated by firewalls already. We conclude that brute force enumeration attacks are infeasible with BridgeSPA.

5.1.1 Other Aliveness Checks

BridgeSPA actively mitigates an adversary's ability to probe aliveness from a Tor bridge. Since the bridge aliveness attack targets bridges on home computers, a Tor bridge may not be the only software running that can demonstrate aliveness. For example, firewalls can be configured differently with respect to external connection requests to closed local ports. That is, firewalls may silently drop packets, which is consistent with BridgeSPA's behaviour when an SPA authorization fails, or send a connection reset ("RST") packet. A bridge machine's firewall should be configured such that it cannot be easily prompted by an adversary to send a RST packet and demonstrate aliveness. Any open port on the machine could similarly provide aliveness information to an adversary. Other publicly accessible services running on the machine could be concealed by traditional SPA solutions. Ideally, BridgeSPA would provide recommendations regarding system configuration or other software that could be probed for aliveness.

5.2 Bridge Client Detection Attacks

We divide client attacks into those that a passive adversary might perform, and those that an active adversary can perform. In both of these cases the adversary is attempting to determine whether a client is attempting to use a Tor bridge.

5.2.1 Active Adversaries

We first consider what a powerful, active adversary is capable of, especially if she suspects that a client may be connecting to a Tor bridge.

As previously mentioned, bridge hoarding could still reveal a particular host as a Tor bridge. An adversary who observes a client connect to a host that has been previously listed as a Tor bridge would certainly become suspicious. She could not, however, easily confirm her suspicions by connecting to the bridge unless she also has a fresh bridge descriptor. In the case of an unlisted bridge, the adversary will never have observed the host listed as a Tor bridge, and it is much less likely that she would be able to obtain a valid MACKey.

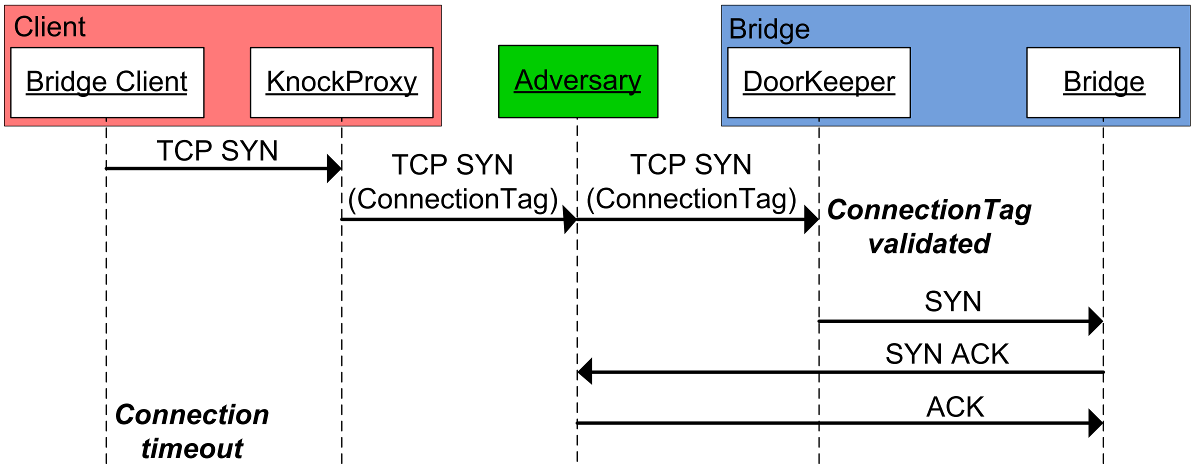


Figure 5.1: A BridgeSPA man-in-the-middle scenario.

An adversary could attempt to use information from the client to connect to the bridge herself. For example, the adversary could replay a previously seen TCP SYN packet from a client to a suspected bridge host. The adversary would need to assume control of the client's IP address, and replay this before the timestamp embedded in the ConnectionTag is stale (90 seconds on average). An even stronger attack is to hijack the TCP connection as it is being synchronized. That is, after the bridge client sends a TCP SYN packet with an embedded ConnectionTag, an active adversary could hijack the connection and attempt to complete the Tor bridge connection. This active man-in-the-middle scenario is illustrated in Figure 5.1.

A way of addressing both of these attacks would be to require that the client sends the entire non-truncated ConnectionTag after the TLS connection is established. Until this is done, the bridge could simply act like some other service that can run on top of TLS (e.g., IMAP). This is similar to previous proposals [13,26], as discussed in Chapter 1. This modification to BridgeSPA is illustrated in Figure 5.2. We note however that currently the TLS certificates used by Tor relays and bridges are distinguishable from other types of TLS certificates. It is impossible for a bridge to convincingly masquerade as another service unless this is addressed. The Tor Project is actively working to address this [3]. We also note that a TCP connection hijack attack is, in general, non-trivial to carry out in practice.

An adversary who unsuccessfully replays a past ConnectionTag might infer that some type of innocuous SPA has taken place. It would be difficult for the adversary, however,

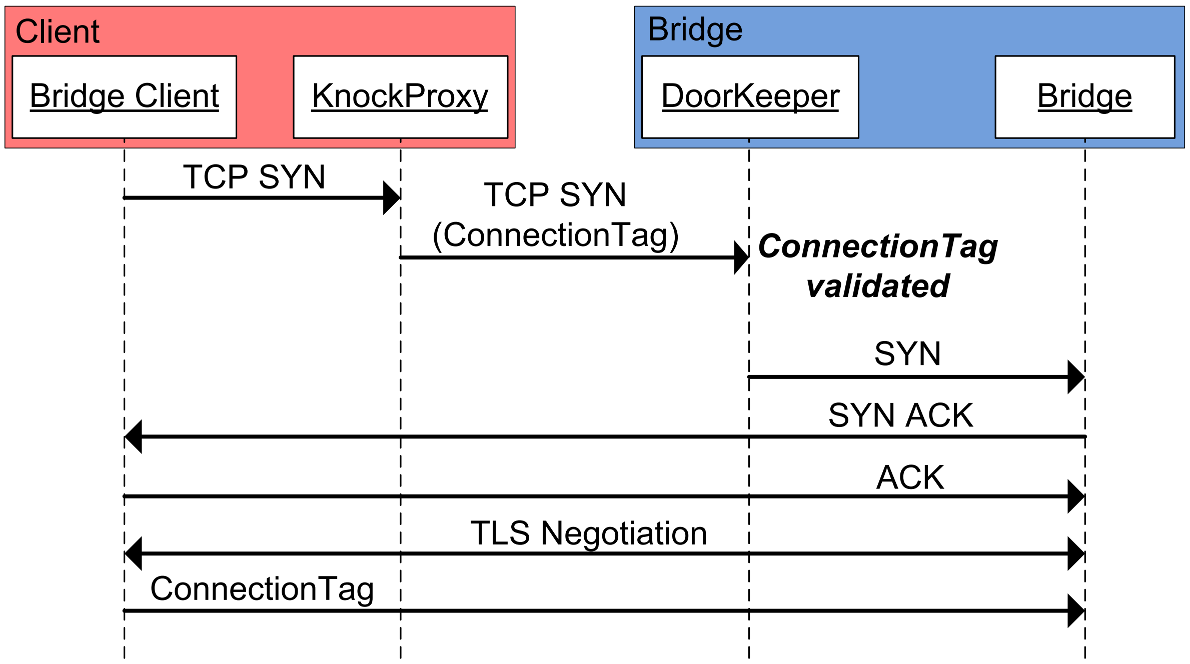


Figure 5.2: BridgeSPA modified to prevent man-in-the-middle.

to distinguish this scenario from a scenario where the target host runs a dynamic firewall whose behaviour may change based on rules unknown to her.

An adversary who suspects that clients are using BridgeSPA could also modify the sequence numbers on all packets in order to prevent SPA from succeeding for the clients whom she suspects. She would also need to similarly change the sequence acknowledgement field on packets inbound to these clients in order to avoid violating the rules of the TCP protocol. In this case, BridgeSPA clients could still evade conclusive detection. If any BridgeSPA connection fails, the client should simply discard that descriptor without any further attempts. This would make it difficult for an adversary to determine if the ISN modification caused the BridgeSPA authorization to fail, or if the destination server was actually offline. Similarly, the KnockProxy could periodically generate TCP SYN packets with invalid ConnectionTags (i.e. when the client is not using that particular bridge) to serve as misinformation to the observer.

We also note that changes to the sequence number of packets *always* break some IP extensions, such as IPSec [14]. Active modification of all packets at line speed in a non-trivial manner seems to be beyond the capabilities of most large active firewalls. For example, the Great Firewall of China examines the contents of packets but does not modify any packets in flight [5].

5.2.2 Passive Adversaries

We also consider what a weaker, passive adversary could do to detect the use of BridgeSPA. By simply observing the BridgeSPA protocol, an adversary can learn only that a client's connection timed out or that he established a TLS connection. This is the same as an adversary observing a bridge client connection today. With the standard implementations of BridgeSPA, the adversary could also conclude that the ISN distribution is consistent with a Linux 2.6 system. Chapter 6 analyzes the potential to use covert channels that are consistent with Microsoft Windows. Ideally the KnockProxy should be able to choose covert channels that are consistent with any client's TCP/IP stack, while still being detectable by the DoorKeeper.

Currently, there exists a race condition between TCP connections created by BridgeSPA and other TCP connections. Specifically, BridgeSPA currently does not observe other outgoing TCP SYN packets to strictly ensure that the TCP timestamp for its connections will be ordered correctly with respect to others. If a passive adversary was sufficiently close to the client and observed a TCP timestamp anomaly, they may suspect that a BridgeSPA-like tool is being used.

As mentioned earlier, Tor traffic is distinguishable from, for example, HTTPS. A passive adversary may still recognize a flow of traffic as Tor-like traffic, but BridgeSPA does not help with this detection.

If an adversary notices a statistically significant delay in responses when a client connects to a particular host, compared to hosts in the same network, he may try to infer that the destination host is running some form of SPA. Vasserman et al. [36] consider the delay introduced by the SilentKnock daemon that runs on the server, `sknockd`, the equivalent of the BridgeSPA DoorKeeper. From their measurements of the userspace version of `sknockd`, they consider an adversary who is several hops away and has perfect knowledge of what the expected non-SPA timing should be. They conclude the adversary would need to observe hundreds of successful connections before gaining an advantage in distinguishing between whether the destination host is running SilentKnock or simply a dynamic firewall.

We similarly measured the timing of our proof-of-concept implementation of BridgeSPA. Table 5.1 shows the difference in timing of SYN and SYN ACK packets from a client connecting to a basic echo server when the server is running behind the DoorKeeper

Table 5.1: A passive listener measuring the difference in timing between 5000 sets of SYN and SYN ACK packets between two other hosts on the same physical network hub. We consider cases when one machine connects to an echo server running on the other machine using the desktop versions of BridgeSPA KnockProxy and DoorKeeper, and also with no BridgeSPA components. The values below are averaged from the 5000 sets. The echo server machine was Thinkpad X60 with a 1.6 GHz Core 2 Duo processor and 4 GiB of RAM.

	SYN/SYN ACK difference ± stddev
Without BridgeSPA	280 ± 20 μs
With BridgeSPA	370 ± 80 μs

compared to when it is not. The client and server were connected to the same network hub as the eavesdropping machine that performed the network measurements.

The measurements show that the DoorKeeper introduces only a small amount of delay, less than 100 μs on average. If the adversary is comparing observed BridgeSPA connection times to the connection times of clients to similar hosts, we, like SilentKnock [36], believe the timings would be inconclusive in detecting the use of BridgeSPA unless the adversary is sufficiently close to this network and has collected a large amount of data. While we believe our implementation can be further optimized, the improvements described in section 7.2 will also add a small amount of overhead for the DoorKeeper.

As shown in Figure 5.3, our experiment measures the difference an adversary could observe in the worst-case scenario. In a more realistic scenario, higher latencies and more variance would make it much more difficult for an adversary to observe a delay introduced by the DoorKeeper.

To demonstrate the difficulty an adversary would have detecting the use of BridgeSPA in scenarios with higher latency, we have conducted similar tests using ModelNet [35] to simulate higher latency. ModelNet is a large-scale network emulator that allows users to simulate network topologies with the specified properties. Researchers may test software over these simulated networks usually without modifications. The BridgeSPA desktop implementation did require some modifications to ensure that both the KnockProxy and DoorKeeper use the same values for the source and destination IP addresses and IP ID field when calculating ConnectionTags.

Figure 5.4 shows our ModelNet experimental configuration. Specifically, the KnockProxy and DoorKeeper ran on the same machine but on different simulated networks. The ModelNet host inserts 100 ms delays between the two simulated networks. To simulate a passive observer, we captured outgoing traffic on the ModelNet host.

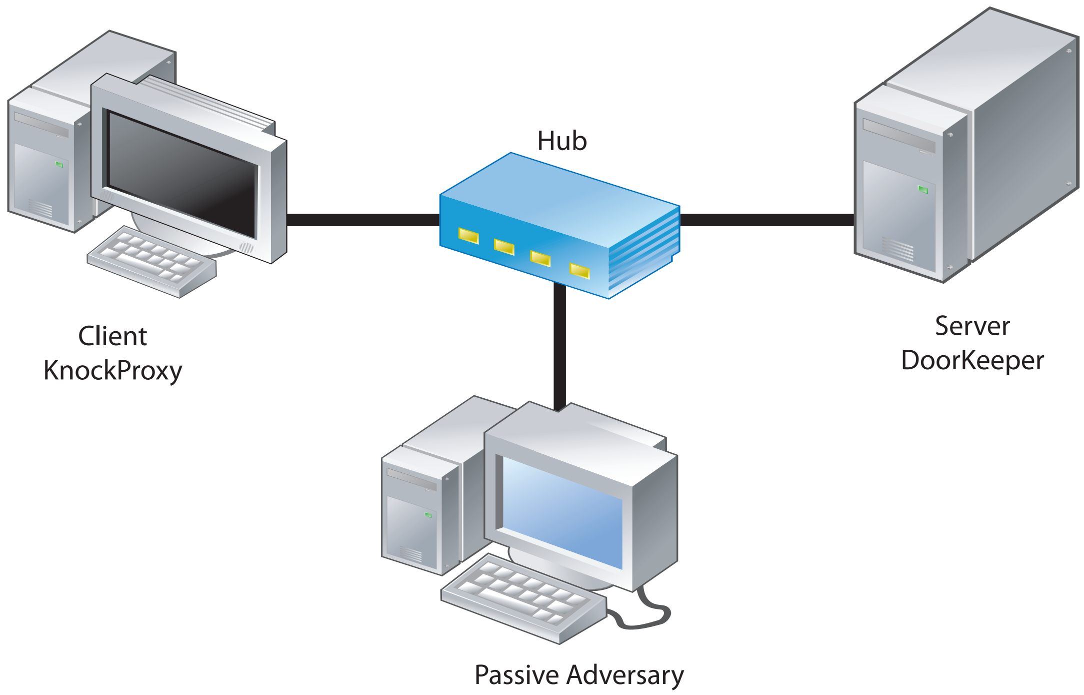


Figure 5.3: The network configuration of the DoorKeeper delay measurement experiment (standard implementation).

The results in Table 5.2 should not be surprising, as they follow closely from the results in Table 5.1. These measurements do not account for additional variance in latency that two Internet-connected peers often observe over time, from changes in congestion in the route that connects them. Despite this allowance the two samples in Table 5.2 are extremely close.

Finally, we also measured the delay introduced by running the OpenWRT version of the DoorKeeper on our test router (described in section 4.3). Since this router is significantly less powerful than the machines used in our other DoorKeeper tests, we expect the DoorKeeper delay to be higher. The configuration for this experiment is pictured in Figure 5.5. Our client is connected to a hub that is also connected to the router and a passive observer. The router also connects to the machine running the service protected by the DoorKeeper, but on a different network interface.

Table 5.3 shows the delay introduced by our test router running the DoorKeeper. We see that the router delays a SYN ACK packet by approximately 800 μs when it is configured

Table 5.2: A passive listener measuring the difference in timing between 2300 sets of SYN and SYN ACK packets between two hosts on separate simulated networks. We consider cases when one machine connects to an echo server running on the other machine using the desktop versions of BridgeSPA KnockProxy and DoorKeeper, and also with no BridgeSPA components. The values below are averaged from the 2300 sets. The machine running the BridgeSPA processes had an Intel E5620 Xeon 2.40 GHz CPU and 8 GiB of RAM.

	SYN/SYN ACK difference ± stddev
Without BridgeSPA	200.4 ± 0.2 ms
With BridgeSPA	200.5 ± 0.3 ms

Table 5.3: A passive listener measuring the difference in timing between 1000 sets of SYN and SYN ACK packets between two hosts. We consider cases when one machine connects to an echo server with and without BridgeSPA components involved. In this case, the client runs the KnockProxy and an intermediary router runs the DoorKeeper. The values below are averaged from the 1000 sets. The test router in this experiment is the same one used in our other router experiments.

	SYN/SYN ACK difference ± stddev
Without BridgeSPA	280 ± 20 μs
With BridgeSPA	1060 ± 60 μs

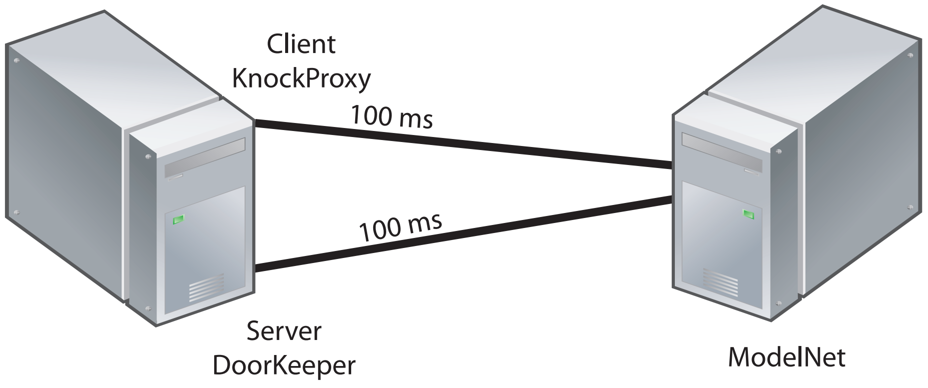


Figure 5.4: The network configuration of the DoorKeeper (standard implementation) delay measurement experiment in a ModelNet simulated network environment.

with BridgeSPA. This is still a reasonable delay in some scenarios, for example when the client and adversary are in a network sufficiently distant from the BridgeSPA router, when the adversary is unlikely to collect many sample timings, or when the adversary has no reference to compare the collected timings with. The DoorKeeper is also likely to show performance improvements if it is ported to run in kernel space.

Related to the attack discussed above, an adversary could also observe differences between connection requests from a client to the bridge port on a particular host and other services that may be running on the same machine. We discussed in section 5.1.1 why a bridge operator who is relying on BridgeSPA to mitigate the bridge aliveness attack should not have other publicly accessible services running.

Another indicator that a client may be connecting to a Tor bridge using BridgeSPA was first observed in Chapter 4. If a client's connection to the bridge has a high throughput capacity, we observed that the connection speed can be affected by the KnockProxy. We observed this effect particularly in our OpenWRT experiments.

In 2009, Perry [25] measured the average stream transfer speed for Tor nodes to be below 100 KiB/s. In our OpenWRT experiments, the KnockProxy introduced no measurable throughput degradation at 600 KiB/s, but increased the throughput's variance slightly.

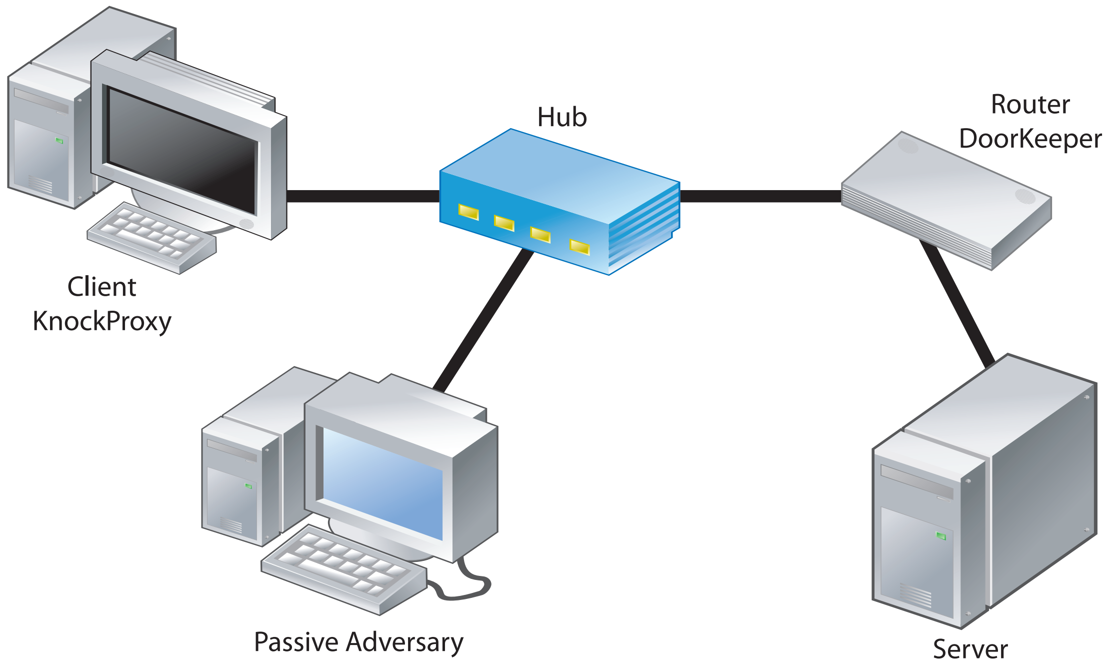


Figure 5.5: The network configuration of the DoorKeeper delay measurement experiment (OpenWRT implementation).

The KnockProxy introduced only a minor speed degradation at 1536 KiB/s. The current KnockProxy is sufficient for today's Tor stream speeds, but there has also been a large body of research looking at ways to increase the speed of Tor [10, 33].

While we do not want to rely on Tor being slow in order for this type of attack to be infeasible, we observe that the processing capabilities of home routers could continue to increase over time. The popular Linksys WRT54G released in 2002 had a 125 MHz processor and 16 MiB RAM. By 2004, Linksys released the WRT54GS with a 200 MHz processor and 32 MiB of RAM. Our test router, a Buffalo WZR-HP-G300NH2 released in 2011, has a 400 MHz CPU and 64 MiB of RAM. As noted in section 4.3.2, the KnockProxy could be further optimized for devices with limited resources.

Chapter 6

TCP/IP Covert Channels in Microsoft Windows

The described implementations of BridgeSPA encode a truncated ConnectionTag using 3 bytes of the ISN and 1 byte of the TCP timestamp value. These fields were chosen because they are expected to appear uniformly random under Linux 2.6. Microsoft Windows does not include a TCP timestamp by default. Ideally, the KnockProxy would be able to alternatively encode values in a TCP SYN packet such that they are consistent with a Microsoft Windows TCP/IP implementation. The issue is that packets used for the BridgeSPA protocol should not differ from other traffic that may be observed. As previously mentioned, the OpenWRT implementation of KnockProxy will explicitly create new connections that follow standard Linux 2.6 TCP/IP properties when it detects a client request. This is still an issue, however, since it is not typical behaviour for Linux's NAT implementation to add TCP options. For this reason we look at other parts of the TCP and IP headers that can be used to encode data in a way that is consistent with Microsoft Windows behaviour.

6.1 Initial Sequence Number Selection

We examine the distribution of ISN selection of a Windows XP SP3 machine. This was done by collecting approximately 675000 ISNs included in TCP SYN packets to a fixed IP address and port. We visualize the data, and analyze it using a tool intended to test the cryptographic suitability of pseudorandom number generators [31].

Figure 6.1 shows the first 2500 collected ISN values. The data was truncated for this visualization to better highlight any potential areas of unexpected concentration of points.

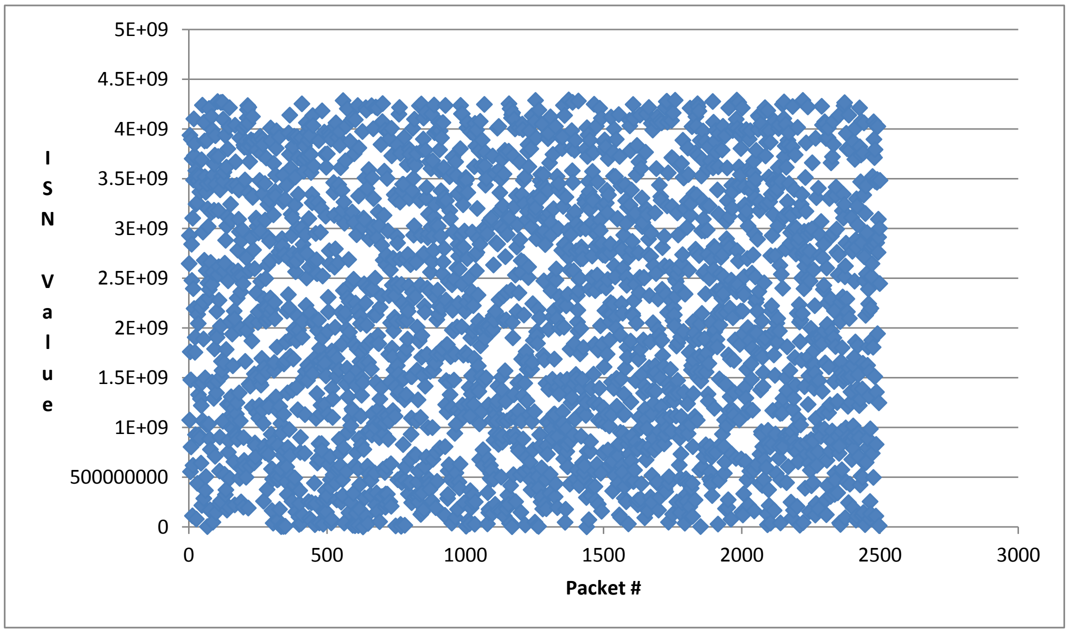


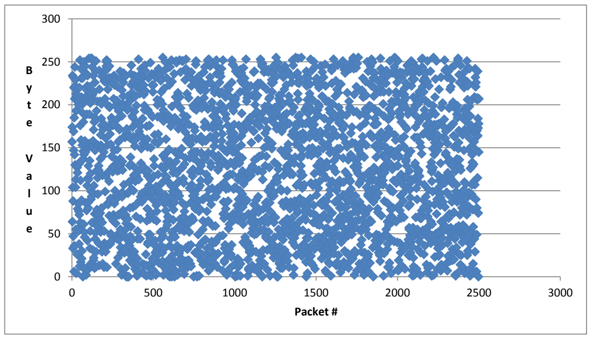
Figure 6.1: A visualization of 2500 initial TCP sequence numbers collected from a Windows XP SP3 machine.

Choosing other random sets of 2500 values shows a very similar picture. All we can conclude by observing this data is that the ISN appears to follow a uniformly random distribution and there is no obvious pattern. Note that the ISN value as graphed will be mostly determined by the most significant byte. Murdoch and Lewis [23] show that the implemented ISN generators in Linux and OpenBSD do not choose the values of all bytes in the same way. For these reason, we have also visualized each byte individually to determine if any non-uniform characteristics are more apparent.

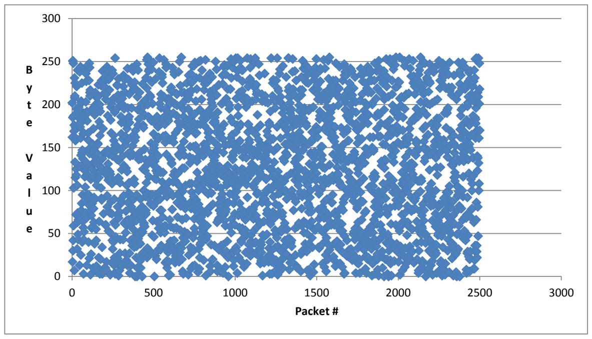
We observe in Figure 6.2 that each individual byte of the collected data also appears to follow a uniformly random distribution.

Next, we analyze the ISN values using a tool [31] developed by the American National Institute of Standards and Technology (NIST). This tool is a test suite intended to determine if a given pseudorandom number generator is inappropriate for cryptographic use. The fifteen tests range in complexity from a basic frequency analysis of the number of zeros and ones in the binary representation of the data, to analyzing the peak heights in the discrete fast Fourier transform of the data.

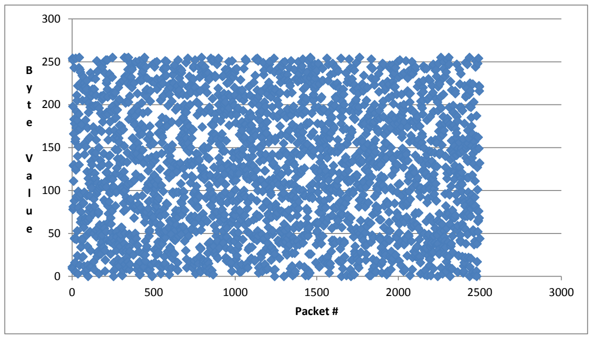
A summary of the results are listed in Table 6.1. The p-value indicates the probabil-



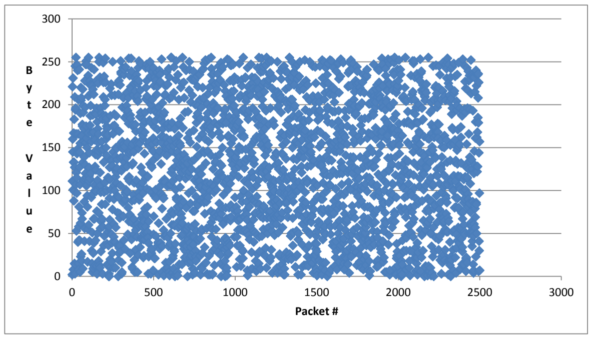
(a) Most significant byte



(b) 2nd most significant byte



(c) 2nd least significant byte



(d) Least significant byte

Figure 6.2: A visualization of 2500 initial TCP sequence numbers collected from a Windows XP SP3 machine, separated by byte.

Table 6.1: The output of the NIST test suite for pseudorandom number generators applied to approximately 675000 ISNs generated by Microsoft Windows.

Test Name	Result	p-value
ApproximateEntropy	SUCCESS	0.147421
BlockFrequency	SUCCESS	0.291548
CumulativeSums	SUCCESS	0.666373, 0.831374
FFT	SUCCESS	0.070172
Frequency	SUCCESS	0.843605
LinearComplexity	SUCCESS	0.728784
LongestRun	SUCCESS	0.362664
NonOverlappingTemplate	SUCCESS	–
OverlappingTemplate	SUCCESS	0.611957
RandomExcursions	SUCCESS	–
RandomExcursionsVariant	SUCCESS	–
Rank	SUCCESS	0.029699
Runs	SUCCESS	0.700467
Serial	SUCCESS	0.348607, 0.251662
Universal	SUCCESS	0.111980

ity that the given test will provide values that are equal or worse randomness than the observed values, under the assumption that the input stream is truly random. The NIST documentation [31] offers an α threshold of 0.01 before declaring the values non-random. That is, if there is less than a 1% chance that a uniformly random generator would generate a stream with random properties, defined by a given test, as poor as the input data, that test fails.

Note that some tests have many variations with distinct p-values. In this case we do not list all the results. We do not explain the purpose or methods of each test, but this documentation is available [31].

As we see in Table 6.1, none of the NIST tests have determined that the Microsoft Windows ISN data stream is non-random. Ultimately there is no way to be sure that the random ISN values are chosen in a way that is cryptographically secure without examining the implementation source code. Since it appears that the numbers are uniformly random, we have modified the KnockProxy and DoorKeeper to use the entire 32-bits of the ISN value to present a ConnectionTag if the timestamp is not present. When using this version, users have to be also be careful to ensure that the TCP and IP packets are otherwise consistent

Table 6.2: The output of the NIST test suite for pseudorandom number generators applied to approximately 675000 ISNs generated by GNU/Linux kernel version 2.6.32.

Test Name	Result	p-value
ApproximateEntropy	SUCCESS	0.022442
BlockFrequency	SUCCESS	0.135554
CumulativeSums	SUCCESS	0.087413, 0.464093
FFT	FAILURE	0.004133
Frequency	SUCCESS	0.282392
LinearComplexity	SUCCESS	0.207229
LongestRun	SUCCESS	0.557603
NonOverlappingTemplate	SUCCESS	–
OverlappingTemplate	SUCCESS	0.053442
RandomExcursions	SUCCESS	–
RandomExcursionsVariant	SUCCESS	–
Rank	FAILURE	0.002708
Runs	SUCCESS	0.489593
Serial	SUCCESS	0.103278, 0.208263
Universal	SUCCESS	0.914637

with an unmodified Microsoft Windows TCP/IP stack. If it is later revealed that Microsoft Windows does not pick ISN values in a way that is uniformly random, BridgeSPA could update the distribution of its encoded values to match.

To demonstrate that the NIST test suite is capable of detecting non-uniformly random data, we also ran the test suite on approximately 675000 ISN values generated by GNU/Linux. The machine used for these tests ran with a kernel version 2.6.32. Since we know that the higher-order byte of the GNU/Linux ISN value is not calculated in a way that is uniformly random [23], we expect some of the NIST tests to fail.

As we see in Table 6.2, two of the NIST tests failed. These tests concluded that the there is less than a 1% chance that a uniformly random byte stream would produce data as non-random as the input data. Most of the other tests also produced lower p-values when compared to the Microsoft Windows test results.

6.2 Other Covert Channels

Another header field that could be used to encode covert channels is the 16-bit IP ID field. Examining the network traces from the experiment above immediately shows us that Microsoft Windows selects this value sequentially. That is, no matter if a host is generating continuous packets to the same machine or single packets to many machines, the IP ID can be predicted. This makes the field inappropriate for use as a covert channel.

Similarly, the 16-bit TCP source port field, when not explicitly specified by an application, is also selected sequentially. This behaviour is also the same between cases when packets are sent by the same machine continuously or single packets to different machines.

Chapter 7

Future Work

Our current implementation is still considered a proof-of-concept. Here we identify parts of BridgeSPA that could be improved in the future.

7.1 Kernel Implementation

The desktop variant of the DoorKeeper and the OpenWRT variant of the KnockProxy could both benefit from kernel implementations to avoid context switches from kernel space to userspace. SilentKnock observed an approximate speed up of 85% in their DoorKeeper kernel implementation compared to their userspace implementation.

7.2 Properly Handling SYN Retransmits

When a client, using the KnockProxy, sends a SYN packet with a ConnectionTag embedded in the ISN and TCP timestamp, there is a chance this packet is lost and must be transmitted. While a re-transmitted SYN will have the same ISN, the TCP timestamp must be different. If the TCP timestamp has the same lower byte as the lost packet, this could reveal information to an adversary who is attempting to detect BridgeSPA usage. Vasserman et al. handle this with SilentKnock [36] by applying a function based on shared knowledge about the non-truncated MACKey to the middle bytes of the timestamp to determine the last byte. Our implementation does not currently handle this, but we believe this behaviour could be properly handled without introducing much extra overhead to the DoorKeeper.

7.3 Properly Handling Finished Connections

The KnockProxy tracks connections by their source and destination IP addresses and ports. The first time it encounters a connection it calculates what the ConnectionTag should be, and stores the offset it needs to apply to the sequence number and TCP timestamp. As packets subsequently arrive and leave the system, the KnockProxy detects that the packets belong to an existing connection and simply locates and applies the offsets. Currently, the KnockProxy does not track when connections end, so if a port is re-used for a different connection a new ConnectionTag will not be calculated.

The solution unfortunately is not as simple as monitoring for TCP FIN and RST flags. While this does handle many cases where a TCP connection ends, if a client sends a TCP SYN and does not get a response, after re-attempting a limited number of times the connection will simply fail without generating additional packets that can be monitored by the KnockProxy. A solution could be for the KnockProxy to also monitor whether a connection has been fully established, and to remove ConnectionTag offsets calculated for ones that have not been established after some period of time. We would need to carefully test that all possible ways a TCP connection can be terminated are properly handled.

7.4 Bridge Authority and BridgeDB Changes

As mentioned in Chapter 4, changes are required in the bridge authority and BridgeDB code bases to support BridgeSPA. For example, currently a bridge authority will accept bridge fingerprints and return the corresponding bridge descriptors. This is contradictory to the introduction of a time-limited MACKey. To introduce BridgeSPA, this particular behaviour of bridge fingerprints would need to be changed. We do not expect our changes to add significantly increased complexity to these components.

7.5 Multiple NAT Layers

We illustrated how the OpenWRT implementation allows clients who are behind a NAT to use BridgeSPA. This is still limited to a single layer of NAT, where the BridgeSPA-enabled OpenWRT router takes the place of the NAT router. If a client's ISP has deployed its own NAT, the KnockProxy and DoorKeeper would use different values for the header data that is used when calculating ConnectionTags. The result is that the KnockProxy would not be able to calculate ConnectionTags in a way that would be successfully validated by the DoorKeeper.

To accommodate this issue, the KnockProxy and BridgeSPA could support a mode that is compatible with multiple layers of NAT, but is weaker against replay threats. The client would need to indicate to the KnockProxy his current public IP, and the KnockProxy would not be able to include the client's source port as a part of the header data that is used when calculating ConnectionTags. Other clients who have the same public IP could still replay an observed ConnectionTag before it expires. To mitigate this, the DoorKeeper could accept a specific ConnectionTag only once within a 10 minute interval. Alternatively, the changes described is section 5.2.1 to mitigate TCP connection hijacking would also be applicable to this type of replay.

7.6 Pluggable Transports

There is a proposal for The Tor Project to support pluggable transports [2], which are custom SOCKS proxies defined on a bridge-by-bridge basis to provide more opportunities to conceal bridge traffic. For example, a bridge could require that clients run a proxy that encapsulates regular Tor TLS traffic in specific HTTP messages. This way bridge traffic could be disguised to appear as if a client is simply interacting with a web site. If a bridge requires a special proxy, this information would be included in the distributed bridge descriptor.

Not only is this goal complementary with BridgeSPA, BridgeSPA could be implemented as a transport plugin. This alternative design would facilitate configuration for clients. Furthermore, the proposed changes to bridge descriptors to support pluggable transports will support the distribution of a BridgeSPA MACKey. BridgeSPA could chain with other transport plugins, but only those that use TCP.

7.7 Privacy and TCP/IP Varieties

Consider an adversary who is capable of monitoring traffic on a household Internet connection. This adversary is also aware that the connection is used by several people, who use their own devices. By using characteristics of the packets he observes, the adversary may be able to classify different connections by their originating operating system. If there is sufficient diversity among operating systems in this household, the adversary can distinguish the connections from the residents without even looking at the content.

Consider a situation where all but one resident has the same TCP/IP characteristics. If our adversary is aware of this distribution, it is easy to see that the unique resident has strictly less privacy.

We have already discussed some of the characteristics of TCP/IP packets that vary between implementations (i.e., chosen distribution of ISN, inclusion of TCP options). In an experiment in 2003 conducted by Lippmann et al. [18], they were able to passively detect the operating systems of nodes generating traffic with an error rate of approximately 10%. They used 10 packet characteristics such as the TCP maximum segment size (MSS) value and IP time to live (TTL) value.

Since the OpenWRT version of the KnockProxy locally initializes new connections for all outgoing connections, all clients behind the router will have the same TCP/IP characteristics when observed from the Internet. In this case, an adversary would not be able to distinguish connections made from different devices on a shared Internet connection. This property of the KnockProxy could be extended to allow configurable TCP/IP characteristics.

Chapter 8

Conclusion

Tor bridges are intended to help users in censored regimes, but they are easy to detect and block. Worse, if a Tor user opts to serve as a bridge, an adversary can deanonymize a bridge operator's pseudonymous online activities due to the fact that a bridge will always serve clients while its operator is using Tor. BridgeSPA mitigates these issues by making it difficult to detect bridges and test whether they are online. BridgeSPA is based on an SPA scheme that was proven undetectable by previous work. Our protocol is resilient to many attacks. We provide a working proof-of-concept that targets GNU/Linux machines, and another for OpenWRT routers.

References

[1] Jacob Appelbaum. Port Knocking for Bridge Scanning Resistance. https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/ideas/xxx-port-knocking.txt, April 2009. [Online; accessed June 2011].

[2] Jacob Appelbaum and Nick Mathewson. Pluggable transports for circumvention. https://gitweb.torproject.org/torspec.git/blob_plain/HEAD:/proposals/180-pluggable-transport.txt, October 2010. [Online; accessed June 2011].

[3] Jacob Appelbaum and Gladys Shufflebottom. Draft spec for TLS certificate and handshake normalization. https://gitweb.torproject.org/torspec.git/blob_plain/HEAD:/proposals/179-TLS-cert-and-parameter-normalization.txt, February 2011. [Online; accessed June 2011].

[4] Paul Barham, Steven Hand, Rebecca Isaacs, Paul Jardetzky, Richard Mortier, and Timothy Roscoe. Techniques for lightweight concealment and authentication in IP networks. *Intel Research Berkeley. July*, 2002.

[5] Richard Clayton, Steven J. Murdoch, and Robert Watson. Ignoring the Great Firewall of China. In *Privacy Enhancing Technologies*, pages 20–35. Springer, 2006.

[6] Roger Dingledine. Behavior for bridge users, bridge relays, and bridge authorities. https://gitweb.torproject.org/torspec.git/blob_plain/HEAD:/proposals/125-bridges.txt, November 2007. [Online; accessed July 2011].

[7] Roger Dingledine. Re: Guard nodes. http://archives.seul.org/or/dev/Jan-2008/msg00011.htm, January 2008. [Online; accessed July 2011].

[8] Roger Dingledine. Iran blocks Tor; Tor releases same-day fix. https://blog.torproject.org/blog/iran-blocks-tor-tor-releases-same-day-fix, September 2011. [Online; accessed November 2011].

[9] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium*. USENIX Association, 2004.

[10] Roger Dingledine and Steven J. Murdoch. Performance Improvements on Tor or, Why Tor is slow and what were going to do about it. `http://www.torproject.org/press/presskit/2009-03-11-performance.pdf`, 2009. [Online; accessed November 2011].

[11] John Giffin, Rachel Greenstadt, Peter Litwack, and Richard Tibbetts. Covert messaging through TCP timestamps. In *Privacy Enhancing Technologies*, pages 194–208. Springer, 2002.

[12] Eu-Jin Goh, Dan Boneh, Benny Pinkas, and Philippe Golle. The Design and Implementation of Protocol-Based Hidden Key Recovery. *Information Security*, pages 165–179, 2003.

[13] George Kadianakis. Re: Proposal 176: Proposed version-3 link handshake for Tor. `http://archives.seul.org/or/dev/Feb-2011/msg00012.html`, February 2011. [Online; accessed June 2011].

[14] Stephen Kent and Randall Atkinson. RFC2402: IP Authentication Header. *RFC Editor United States*, 1998.

[15] Jan Kneschke. Lighttpd. `http://www.lighttpd.net/`. [Online; accessed November 2011].

[16] Martin Krzywinski. Port knocking: Network authentication across closed ports. *SysAdmin Magazine*, 12(6):12–17, 2003.

[17] Andrew Lewman. China blocking Tor: Round Two. `https://blog.torproject.org/blog/china-blocking-tor-round-two`, March 2010. [Online; accessed June 2011].

[18] Richard Lippmann, David Fried, Keith Piwowarski, and William Streilein. Passive operating system identification from TCP/IP packet headers. In *Workshop on Data Mining for Computer Security*, page 40. ICDM, 2003.

[19] Karsten Loesing and Nick Mathewson. Tor BridgeDB. `https://gitweb.torproject.org/bridgedb.git/blob_plain/HEAD:/bridge-db-spec.txt`. [Online; accessed June 2011].

[20] Moxie Marlinspike. Knockknock. `http://www.thoughtcrime.org/software/knockknock/`, December 2009. [Online; accessed November 2011].

[21] Nick Mathewson. Bridge Guards and other anti-enumeration defenses. `https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/188-bridge-guards.txt`, October 2011. [Online; accessed November 2011].

[22] Jon McLachlan and Nick Hopper. On the risks of serving whenever you surf: vulnerabilities in Tor's blocking resistance design. In *Proceedings of the 8th ACM Workshop on Privacy in the Electronic Society*, pages 31–40. ACM, 2009.

[23] Steven J. Murdoch and Stephen Lewis. Embedding covert channels into TCP/IP. In *Information Hiding*, pages 247–261. Springer, 2005.

[24] Ookla. Net Index Internet Data. `http://www.netindex.com/source-data/`. [Online; accessed November 2011].

[25] Mike Perry. TorFlow: Tor Network Analysis. *HotPETS*, 2009.

[26] Shane Pope. Port-Scanning Resistance in Tor Anonymity Network. Honours thesis, University of Texas at Austin, December 2009.

[27] The Tor Project. Tor Metrics Portal. `http://metrics.torproject.org`. [Online; accessed June 2011].

[28] Michael Rash. fwknop: Single Packet Authorization and Port Knocking. `http://cipherdyne.org/fwknop/`. [Online; accessed November 2011].

[29] Michael Rash. Single packet authorization with fwknop. *login: The USENIX Magazine*, 31(1):63–69, 2006.

[30] Craig H. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday*, 2(5), 1997.

[31] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, et al. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. *Reports on Computer Systems Technology*, 2010.

[32] Rob Smits, Divam Jain, Sarah Pidcock, Ian Goldberg, and Urs Hengartner. BridgeSPA: Improving Tor Bridges with Single Packet Authorization. In *Proceedings of the 10th annual ACM Workshop on Privacy in the Electronic Society*, pages 93–102. ACM, 2011.

[33] Can Tang and Ian Goldberg. An improved algorithm for Tor circuit scheduling. In *Proceedings of the 17th ACM conference on Computer and Communication Security*, pages 329–339. ACM, 2010.

[34] The Netfilter Core Team. The netfilter.org “libnetfilter_queue” project. `http://www.netfilter.org/projects/libnetfilter_queue/index.html`. [Online; accessed June 2011].

[35] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.

[36] Eugene Y. Vasserman, Nick Hopper, and James Tyra. SilentKnock: practical, provably undetectable authentication. *International Journal of Information Security*, 8(2):121–135, 2009.