

# SPATor: Improving Tor Bridges with Single Packet Authorization

Rob Smits  
rdfsmit@cs.uwaterloo.ca

Ian Goldberg  
iang@cs.uwaterloo.ca

Divam Jain  
d2jain@cs.uwaterloo.ca

Sarah Pidcock  
snpidcoc@cs.uwaterloo.ca

Urs Hengartner  
uhengart@cs.uwaterloo.ca

Cheriton School of Computer  
Science  
University of Waterloo

## 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. 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, vulnerabilities in Tor's bridge implementation make it easy to discover large numbers of bridges. 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 SPATor as a method to mitigate this issue. A client using SPATor 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 SPATor as a working proof-of-concept, which is available under an open-source licence.

## Categories and Subject Descriptors

C.2.0 [Computer-Communication Networks]: General—*Security and Protection*

## General Terms

Security, Design

## Keywords

Privacy, Anonymity, Tor, Blocking Resistance, Port Knocking

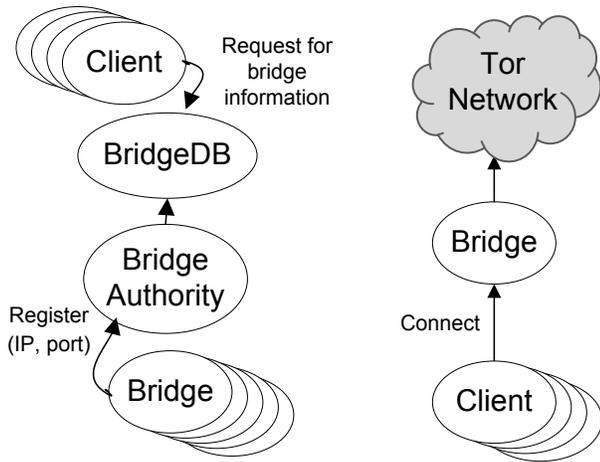
## 1. INTRODUCTION

Tor [13] is an open-source low-latency anonymity network that sees approximately 250,000 users per day [4]. Tor relies on volunteers to operate *relays* that forward end users' traffic. Tor may be used to defeat some forms of Internet censorship by allowing users to connect to censored web-sites indirectly via a series of relays and encrypted tunnels

between them. However, as a list of current Tor relays is easily retrievable from centralized, publicly known *directory authorities*, it is trivial for an Internet access provider to block all connections to Tor by blocking access to IP addresses of Tor relays. As of June 2011, China blocks Tor via this method [19].

In an attempt to mitigate the ease of blocking Tor, unlisted Tor 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. The standard Tor client may be easily configured to operate as a bridge. 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. As shown in Figure 1, the bridge authority keeps track of valid bridges, and the BridgeDB [3] provides the mechanisms for distributing bridge information through the web and by e-mail. The mechanism for distributing bridge information to clients aims to make it easy for any client to find a few bridges through the web or via e-mail. At the same time it attempts to make it difficult for someone to find many bridges in a short amount of time. This is done by restricting the distribution of bridge descriptors to one set per 24-bit IP address prefix in a week. However, the distribution mechanism does not account for attacks in which a user can gain control of many IP addresses through open proxies or botnets. Also, a relay operator could discover many bridges by attempting a bridge connection to any non-relay node that connects to it.

McLachlan et al. [20] have identified 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. They described the attack as three unique phases, which we collectively refer to as the “bridge aliveness attack”:



**Figure 1: Basic interaction among bridge-related entities.**

1. The **bridge discovery phase** involves collecting a large list of bridge addresses and descriptors for use later in the attack.
2. The **winnowing phase** narrows down the above list of bridges that could have possibly contributed material to a website with user-generated content under a particular pseudonym by correlating timestamps of pseudonymous posts with bridge “aliveness” monitored through frequent polling of known bridges.
3. The **confirmation phase** is performed using a combination of circuit clogging and timing attacks to confirm the suspected source of the pseudonymous contribution on the website. To achieve this, the attacker must be able to embed or control some content on this page. The result is that the contributor’s IP address, which he believed was hidden by Tor, is revealed to the attacker.

Apart from compromising the bridge operator’s anonymity, the capacity to discover a large number of live bridges also makes it feasible to perform real-time blocking of many currently live bridges. This is an undesirable situation from the standpoint of maintaining bridge availability. Furthermore, the attack also discourages users from running bridges if there is a chance their anonymity can be compromised.

McLachlan et al. [20] 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 an entry 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 since the hash does not change unless the bridge’s public key changes, and hoarding would still be possible.

We also note that, because bridge users don’t use entry guards, some Tor relays can infer that a connection is from a bridge. That is, if a Tor relay is not an entry guard and receives a connection from an IP address that is not listed as a Tor relay, it may conclude that the IP address belongs to a bridge. This issue has been previously discussed [11, 12].

## 1.1 Our Contributions

We introduce SPATor, a system that mitigates the risks of serving as a bridge while using Tor by making it more difficult to hoard bridge information and query bridge aliveness. As users request bridge information from the BridgeDB, SPATor requires that an additional per-bridge key is provided to them. This key is in addition to the bridge IP, port, and fingerprint that are currently distributed. The key will only be valid for a time period defined by the associated bridge. Clients using SPATor must prove knowledge of the appropriate key in order to access a bridge or to query aliveness of the bridge. The key is used in an innocuous single packet authorization (SPA) protocol [23, 25], 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 is unable to learn that the protocol is being used to connect to a bridge. The innocuous SPA protocol used in SPATor is based on an existing system, SilentKnock [25], and has been modified for use with bridges so they would not need to maintain explicit per-client counters.

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

In the next section we discuss related work. We formalize our goals and adversarial model in section 3. In section 4 we outline the details of the SPATor protocol. We describe our implementation and consider possible attacks on SPATor in sections 5 and 6, respectively. Finally, we discuss future work in section 7, and conclude in section 8.

## 2. RELATED WORK

The core strategies of SPATor 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 in the target system. A strategy used to limit a port scan’s effectiveness is to drop all packets that do not arrive from a predefined whitelist of IP addresses. However, bridges by definition receive connections from strangers, so the whitelisting strategy cannot be applied in this scenario. The whitelist would also re-

quire logging user IP addresses for clients connecting via bridges, which is an undesirable requirement in anonymity-preserving systems like Tor.

In 2002, Barham et al. [9] 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 a secret key encoded in the TCP and IP headers. While this does 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 being passed onto an application. In 2003, Krzywinski [18] 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. Unlike SAS, port knocking can be implemented in Linux with simple shell scripts interacting with firewall rules. In 2006, Rash [23] described *single packet authorization* (SPA) as a more elegant solution to the same problem. 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. 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 [14, 21, 24]. In 1997, Rowland [24] described how the IP ID field in an IP packet as well as the initial sequence number (ISN) in a TCP packet can be used to encode information. This work was later strengthened by Giffin et al. [14] to use the TCP timestamp field, and also to use message authentication codes to encode the covert messages. Murdoch and Lewis [21] 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. [15] 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 [25], a form of SPA and SAS that takes into consideration the aforementioned work on TCP/IP covert channels. They also present a formal model against which we can 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 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/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.

SPATor uses a form of SPA and SAS based on SilentKnock. SPATor also uses the lower 3 bytes of the ISN value, and the lower byte of the TCP timestamp field. SPATor does not try to maintain a counter for each client to prevent replay, but instead includes loosely rounded UTC time in the MAC pre-image. 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 section 5. For convenience, the structure of a TCP/IP packet header as used in SPATor is shown in Tables 1 and 2.

Specific to Tor, there have been investigations [16, 22] 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 secret is not sent to the bridge, 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 in the winnowing phase, but we describe how this method may be used to protect against hijacking attacks on the client side in section 6.2.

Similar to the suggestions for Tor discussed above, there has been a proposal to use port knocking and SPA for Tor bridges to make them more resilient against detection [6]. 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 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.

### 3. GOALS AND ADVERSARIAL MODEL

We have two major goals. As suggested above, our first goal is to mitigate the bridge aliveness attack. Specifically, we wish to remove the ability for an attacker to easily query a bridge’s aliveness unless he has recently obtained the bridge’s key. This will significantly increase the amount of network resources an attacker must have to carry out the bridge aliveness attack, since hoarded bridge information rapidly becomes stale.

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 [8]. Our work should not hinder their efforts in any way. A passive 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.

**Table 1: IP packet header.** We use *italics* for values that can be used for covert channels, as described by Murdoch et al. [21].

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																			
<i>Identification (IP ID)</i>								<i>Flags</i>				<i>Fragment Offset</i>																			
Time to Live				Protocol				Header Checksum																							
Source Address																															
Destination Address																															
<i>Options</i>												Padding																			

**Table 2: TCP packet header.** We use *italics* for values that can be used for covert channels, as described by Murdoch et al. [21]. Values that are in **bold** are used by SPATor.

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
<i>Source Port</i>								Destination Port																							
<b>Sequence Number</b>																															
Acknowledgement Number																															
Data Offset				Reserved				TCP Flags [CUAPRSF]				Window Size																			
Checksum								Urgent pointer																							
<i>Options and Timestamp</i>												Padding																			

Our 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 avoid degrading service by forcing a bridge to probabilistically stop serving clients when it otherwise would be capable of doing 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 aliveness attack, we consider a similar adversarial model to the one employed by McLachlan et al. [20] when they presented the problem. Specifically, 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 and 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.

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.

## 4. SPATor PROTOCOL

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

The SPATor protocol utilizes a pre-shared key to generate message authentication codes (MACs) that are used to determine the legitimacy of a bridge connection request. We consider three main phases of a bridge’s life cycle that are affected by SPATor: bridge registration, bridge request and client connection. An outline of changes made to the bridge’s life cycle is illustrated in Figure 2.

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

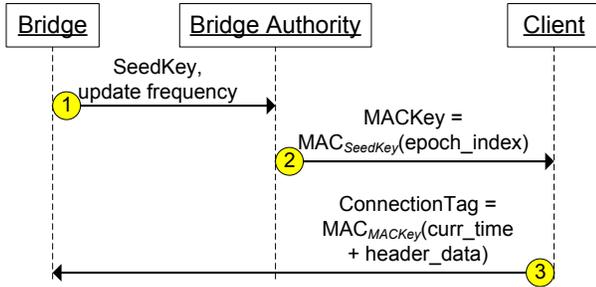


Figure 2: Changes to Tor bridge-related communication in SPATor. 1. Upon registration with the bridge authority, a bridge includes a `SeedKey` and update frequency. 2. The bridge authority distributes bridge information along with the current `MACKey`, derived from the `SeedKey`. This information typically 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.

for connecting to it. This includes the IP address, port, and optionally, a fingerprint. In SPATor, the `SeedKey` (a 256-bit random value) and an associated update frequency are also included. A reasonable value for an update frequency would be between 1–7 days. 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 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` which will be 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. This is outlined in Figure 3. To gracefully deal with edge cases, the `DoorKeeper` also calculates and compares `ConnectionTags` of the previous and next minutes before dropping a packet.

The client embeds a `ConnectionTag` using the TCP SYN packet as a covert channel. As mentioned above, following a strategy based on `SilentKnock` [25], to a passive observer

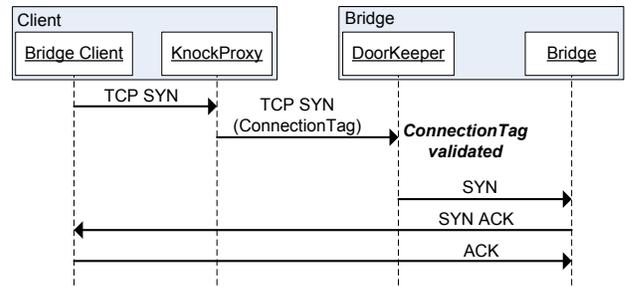


Figure 3: A client using SPATor to connect to a Tor bridge.

the request is indistinguishable from a connection request not containing authorization information.

## 5. IMPLEMENTATION

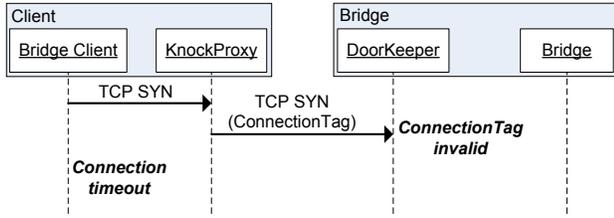
We have developed a proof-of-concept implementation of SPATor to test our protocol.<sup>1</sup> Our implementation requires no changes to the Tor client or bridge software. To support a deployed instance of SPATor, however, the Tor bridge authority and `BridgeDB` would need to be modified. The implementation targets Linux, with kernel version 2.6.4 or later due to our reliance on the `libnetfilter_queue` library [2]. This library provides a user-space API for manipulating network packets. Our implementations of both the `DoorKeeper` and `KnockProxy` use `libnetfilter_queue` to implement their respective parts of the SPATor protocol. On the client side we also developed a small tool to assist with configuration. This tool will take a SPATor bridge descriptor, configure Tor to use the bridge, and pass along the information to the running `KnockProxy` process.

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`. All subsequent packets sent to and received from this bridge will also be modified to have their sequence and acknowledgment numbers adjusted appropriately.

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` 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`. The packet

<sup>1</sup>Available at <http://crisp.uwaterloo.ca/software/> under an open-source licence.



**Figure 4: A failed authorization scenario with SPATor.**

is allowed to continue if this check succeeds. If the check fails, the packet is rejected and will not be processed by the operating system’s TCP stack.

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. 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.

## 5.1 Unlisted Bridges

In section 4, we described how a client who received bridge information from The Tor Project may connect to a bridge using the SPATor 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 compatible with our proof-of-concept implementation.

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 SPATor protocol. Both of these are possible with our proof-of-concept implementation. 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 client IP in the computation of the MACKey, for example, would suffice.

## 6. ATTACKS ON SPATOR

In this section, we analyze SPATor’s effectiveness by discussing some attacks that are possible under the adversarial model defined in section 3.1.

### 6.1 Bridge Aliveness Attacks

We can attempt to re-apply the bridge aliveness attack from our introduction. In the bridge acquisition 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. We consider this to be a substantial improvement.

A determined adversary who thinks that a particular bridge is concealed by SPATor could try to determine aliveness by guessing the correct ConnectionTag. As mentioned in section 4, a DoorKeeper will accept three ConnectionTags at any time. As a ConnectionTag is 32 bits long, this leaves an attacker with an expected  $2^{31}/3$  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 SPATor.

In section 7.4, we describe aliveness attacks on bridge machines that do not target the running bridge software.

### 6.2 Bridge Client Detection Attacks

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

#### 6.2.1 Passive Adversaries

By simply observing the SPATor 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 our current implementation, the adversary could also conclude that the ISN distribution is consistent with a Linux 2.6 system. Since our proof-of-concept implementation only operates with Linux 2.6 clients this is not an issue. As support for more operating systems is added in the future, the ISN distribution for a client using the KnockProxy should continue to match the ISN distribution for connections that do not.

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 SPATor does not help with this detection.

#### 6.2.2 Active Adversaries

We also consider what an 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 bridge would certainly become suspicious. She could not, however, easily confirm her suspicions by connecting to the bridge unless she has a fresh bridge descriptor.

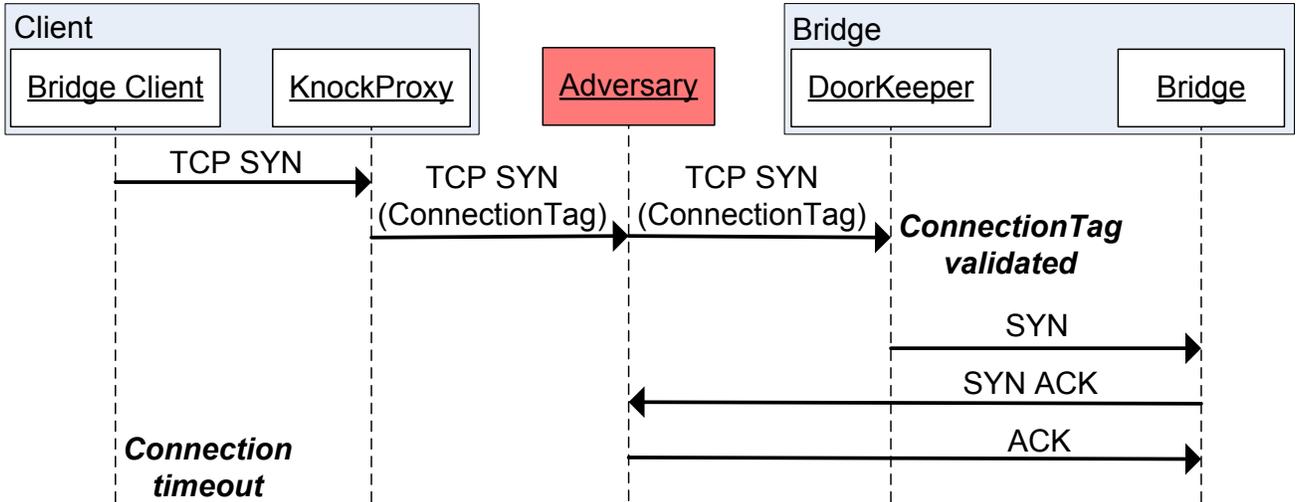


Figure 5: A SPATor man-in-the-middle scenario.

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. 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 man-in-the-middle scenario is illustrated in Figure 5.

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 [16, 22], as discussed in section 2. This modification to SPATor is illustrated in Figure 6. 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.

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, 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. Furthermore, an adversary could try to monitor timing differences when a client connects to a particular host versus the time to connect to hosts in similar IP ranges. If an adversary notices a statistically significant delay in responses when a client connects to a particular host, he may try to infer that the destination host is running some form of SPA. Vasserman et al. [25] consider the delay introduced by the SilentKnock daemon that runs on the server (`sknockd`). From their measurements of the

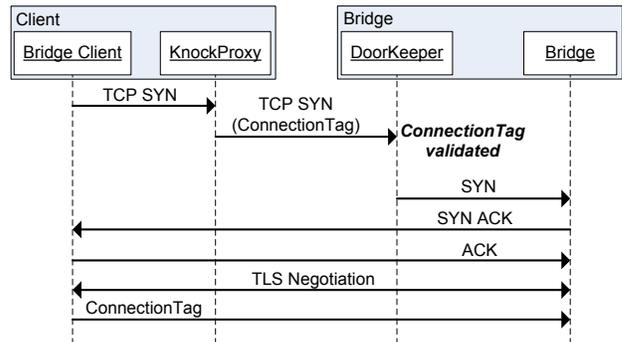


Figure 6: SPATor modified to prevent man-in-the-middle.

user space 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 SPATor. Table 3 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 compared to when it is not. The client and server were connected to the same network hub as the machine that performed the network measurements. This configuration is the best-case scenario for an adversary who is attempting to detect the use of SPATor. The measurements show that the DoorKeeper introduces only a small amount of delay, less than one hundred microseconds on average. If the adversary is comparing observed SPATor connection times to the connection times of clients to similar hosts, we, like SilentKnock [25], believe the timings would be inconclusive in detecting the use of SPATor unless the adversary is sufficiently close to this net-

**Table 3: 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 SPATor KnockProxy and DoorKeeper, and also with no SPATor components. The values below are averaged from the 5000 sets. The echo server machine was Thinkpad X60 with a 1.6GHz Core 2 Duo processor and 4GB of RAM.**

	SYN/SYN ACK difference $\pm$ stddev
Without SPATor	280 $\pm$ 20 $\mu$ s
With SPATor	370 $\pm$ 80 $\mu$ s

work 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.

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 discuss in section 7.4 why a bridge operator who is relying on SPATor to mitigate the bridge aliveness attack should not have other publicly accessible services running.

An adversary who suspects that clients are using SPATor 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. We note however that changes to the sequence number of packets *always* break some IP extensions, such as IPSec [17]. 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 [10].

## 7. FUTURE WORK

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

### 7.1 Compatibility, NAT

We have targeted Linux 2.6 due to the availability of its source code and useful libraries. In the future we would ideally target all platforms on which Tor clients and bridges can operate. As an alternative design, we hope to explore the idea of implementing the KnockProxy and DoorKeeper in home routers that are capable of running user-specified firmware. Projects such as Torouter [5] and FreedomBox [1] suggest that this may be a feasible approach. Currently the SPATor protocol will not operate if the client is behind NAT. Implementing KnockProxy and DoorKeeper in home routers may also solve this issue.

### 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 times-

tamp, there is a chance this packet is lost and must be retransmitted. 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 SPATor usage. Vasserman et al. handle this with SilentKnock [25] by applying a function based on shared knowledge about the non-truncated MACKKey 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 Bridge Authority and BridgeDB Changes

As mentioned in section 5, changes are required in the bridge authority and BridgeDB code bases to support SPATor. For example, currently a bridge authority will accept bridge fingerprints and return the corresponding bridge descriptors. This behaviour allows bridge hoarding and would need to be modified. We do not expect our changes to add significantly increased complexity to these components.

### 7.4 Other Aliveness Checks and Recommendations

SPATor 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 SPATor’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, SPATor would provide recommendations regarding system configuration or other software that could be probed for aliveness.

### 7.5 Pluggable Transports

There is a proposal for The Tor Project to support pluggable transports [7], 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 SPATor, SPATor 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 SPATor MACKKey.

## 8. CONCLUSION

Previous work has explored vulnerabilities in the current implementation of Tor bridges. The findings have indicated that with a few trivial assumptions, 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. The SPATor protocol addresses the issues of bridge descriptor hoarding and the bridge aliveness attack while providing the bridge operator with a stronger guarantee that the client received her descriptor from a bridge authority. SPATor is based on a single packet authorization scheme that has been proven to be undetectable by previous work. Our implementation is available under an open-source licence.

## 9. ACKNOWLEDGEMENTS

We would like to thank Ryan Henry and Tariq Elahi as great sounding boards for ideas. We thank NSERC, MITACS, and The Tor Project for funding this work. Finally, we would like to thank Kevin Bauer, Jean-Charles Grégoire, Tao Wang, and Angèle Hamel for reviewing our work along the way.

## 10. REFERENCES

- [1] FreedomBox. <http://wiki.debian.org/FreedomBox>. [Online; accessed June 2011].
- [2] The netfilter.org “libnetfilter\_queue” project. [http://www.netfilter.org/projects/libnetfilter\\_queue/index.html](http://www.netfilter.org/projects/libnetfilter_queue/index.html). [Online; accessed June 2011].
- [3] Tor BridgeDB. <https://gitweb.torproject.org/bridgedb.git/tree>. [Online; accessed June 2011].
- [4] Tor Metrics Portal. <http://metrics.torproject.org>. [Online; accessed June 2011].
- [5] Torrouter. <https://trac.torproject.org/projects/tor/wiki/TheOnionRouter/Torrouter>. [Online; accessed June 2011].
- [6] J. 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].
- [7] J. Appelbaum and N. Mathewson. Pluggable transports for circumvention. [https://gitweb.torproject.org/torspec.git/blob\\_plain/HEAD:/proposals/180-pluggable-transport.txt](https://gitweb.torproject.org/torspec.git/blob_plain/HEAD:/proposals/180-pluggable-transport.txt), October 2010. [Online; accessed June 2011].
- [8] J. Appelbaum and G. 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](https://gitweb.torproject.org/torspec.git/blob_plain/HEAD:/proposals/179-TLS-cert-and-parameter-normalization.txt), February 2011. [Online; accessed June 2011].
- [9] P. Barham, S. Hand, R. Isaacs, P. Jardetzky, R. Mortier, and T. Roscoe. Techniques for lightweight concealment and authentication in IP networks. *Intel Research Berkeley*, July, 2002.
- [10] R. Clayton, S. Murdoch, and R. Watson. Ignoring the Great Firewall of China. In *Privacy Enhancing Technologies*, pages 20–35. Springer-Verlag, 2006.
- [11] R. Dingledine. Behavior for bridge users, bridge relays, and bridge authorities. [https://gitweb.torproject.org/torspec.git/blob\\_plain/HEAD:/proposals/125-bridges.txt](https://gitweb.torproject.org/torspec.git/blob_plain/HEAD:/proposals/125-bridges.txt), November 2007. [Online; accessed July 2011].
- [12] R. Dingledine. Re: Guard nodes. <http://archives.seul.org/or/dev/Jan-2008/msg00011.html>, January 2008. [Online; accessed July 2011].
- [13] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *In Proceedings of the 13th Usenix Security Symposium*, 2004.
- [14] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert messaging through TCP timestamps. In *Proceedings of the 2nd international conference on Privacy enhancing technologies*, pages 194–208. Springer-Verlag, 2002.
- [15] E. Goh, D. Boneh, B. Pinkas, and P. Golle. The Design and Implementation of Protocol-Based Hidden Key Recovery. *Information Security*, pages 165–179, 2003.
- [16] G. 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].
- [17] S. Kent and R. Atkinson. RFC2402: IP Authentication Header. *RFC Editor United States*, 1998.
- [18] M. Krzywinski. Port knocking: Network authentication across closed ports. *SysAdmin Magazine*, 12(6):12–17, 2003.
- [19] A. Lewman. China blocking Tor: Round Two. <https://blog.torproject.org/blog/china-blocking-tor-round-two>, March 2010. [Online; accessed June 2011].
- [20] J. McLachlan and N. 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.
- [21] S. Murdoch and S. Lewis. Embedding covert channels into TCP/IP. In *Information Hiding*, pages 247–261. Springer, 2005.
- [22] S. Pope. Port-Scanning Resistance in Tor Anonymity Network. Honours thesis, University of Texas at Austin, December 2009. [Online; accessed June 2011].
- [23] M. Rash. Single packet authorization with fwknop. *login: The USENIX Magazine*, 31(1):63–69, 2006.
- [24] C. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday*, 2(5), 1997.
- [25] E. Vasserman, N. Hopper, and J. Tyra. Silent Knock: practical, provably undetectable authentication. *International Journal of Information Security*, 8(2):121–135, 2009.