IMUX: Managing Tor Connections from Two to Infinity, and Beyond

John Geddes[†] Rob Jansen[‡] Nicholas Hopper[†]

[†] University of Minnesota
Minneapolis, MN
{geddes, hopper}@cs.umn.edu

[‡] U.S. Naval Research Laboratory
Washington, DC
rob.g.jansen@nrl.navy.mil

ABSTRACT

We consider proposals to improve the performance of the Tor overlay network by increasing the number of connections between relays, such as Torchestra and PCTCP. We introduce a new class of attacks that can apply to these designs, *socket exhaustion*, and show that these attacks are effective against PCTCP. We also describe IMUX, a design that generalizes the principles behind these designs while still mitigating against socket exhaustion attacks. We demonstrate empirically that IMUX resists socket exhaustion while finding that web clients can realize up to 25% increase in performance compared to Torchestra. Finally, we empirically evaluate the interaction between these designs and the recently proposed KIST design, which aims to improve performance by intelligently scheduling kernel socket writes.

Categories and Subject Descriptors

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

Keywords

Tor; anonymity; privacy; performance; attack; socket exhaustion

1. INTRODUCTION

Tor is a widely-used tool for low-latency anonymous and uncensored Internet access, consisting of an overlay network of roughly 5000 relay nodes and an estimated 1 million daily users. In order to provide anonymity, each Tor client periodically chooses three relays and iteratively forms a *circuit* by telescopically contacting each relay through the previously contacted relays, so that each relay knows only two hops in any connection. Because of the high ratio of clients to relays, and because attracting performance-sensitive users can improve the privacy of all users, properly allocating limited bandwidth and processing resources of the relays in a decentralized and privacy-preserving manner is crucial.

One well-recognized performance issue in Tor stems from the fact that all circuits passing between a pair of relays are multiplexed over a single TLS connection. As shown by Reardon and Goldberg [20], this can result in several undesirable effects on performance: a single, high-volume circuit can lead to link congestion, throttling all circuits sharing this link [20]; delays for packet re-transmissions can increase latency for other circuits, leading to "head-of-line" blocking [19]; and long write buffers reduce the effectiveness of application-level scheduling decisions [10].

As a result, several researchers have proposed changes in the transport protocol for the links between relays. Reardon and Goldberg suggested that relays should use a Datagram TLS tunnel at the transport level, while running a separate TCP session at the application level for each circuit [20]; this adds a high degree of complexity (an entire TCP implementation) to the application. Similarly, the "Per-Circuit TCP" (PCTCP) design [5] establishes a TCP session for each circuit, hiding the exact traffic volumes of these sessions by establishing an IPSEC tunnel between each pair of relays; however, kernel-level TCP sessions are an exhaustible resource and we demonstrate in Section 3 that this can lead to attacks on both availability and anonymity. In contrast, the Torchestra transport suggested by Gopal and Heninger [8], has each relay pair share one TLS session for "bulk download" circuits and another for "interactive traffic." Performance then critically depends on the threshold for deciding whether a given circuit is bulk or interactive.

This paper presents a third potential solution, inverse-multiplexed Tor with adaptive channel size (IMUX). In IMUX, each relay pair maintains a set of TLS connections (channel) roughly proportional to the number of "active" circuits between the pair, and all circuits share these TLS connections; the total number of connections per relay is capped. As new circuits are created or old circuits are destroyed, connections are reallocated between channels. This approach allows relays to avoid many of the performance issues associated with the use of a single TCP session: packet losses and buffering on a single connection do not cause delays or blocking on the other connections associated with a channel. At the same time, IMUX can offer performance benefits over Torchestra by avoiding fate sharing among all interactive streams, or per-circuit designs by avoiding the need for TCP handshaking and slow-start on new circuits. Compared to designs that require a user-space TCP implementation, IMUX has significantly reduced implementation complexity, and due to the use of a per-relay connection cap, IMUX can mitigate attacks aimed at exhausting the available TCP sessions at a target relay.

WPES'14, November 03 2014, Scottsdale, AZ, USA
Copyright 2014 ACM 978-1-4503-3148-7/14/11 ...$15.00
http://dx.doi.org/10.1145/2665943.2665948.

Contributions. We make the following contributions to the Tor relay transport literature:

- We describe new socket exhaustion attacks on Tor and PCTCP that can anonymously disable targeted relays, and demonstrate how socket exhaustion leads to reductions in availability, anonymity, and stability.
- We describe IMUX, a novel approach to the circuit-to-socket solution space. Our approach naturally generalizes between the "per-circuit" approaches such as PCTCP and the fixed number of sessions in "vanilla Tor" (1) and Torchestra (2).
- We analyze a variety of scheduling designs for using a variable number of connections per channel through large-scale simulations with the Shadow simulator [11]. We compare IMUX to PCTCP and Torchestra, and suggest parameters for IMUX that empirically outperform both related approaches while avoiding the need for IPSEC and reducing vulnerability to attacks based on TCP session exhaustion.
- We perform the first large scale simulations of the Torchestra design and the first simulations that integrate KIST [10] with Torchestra, PCTCP, and IMUX to compare the performance interactions among the complimentary designs.

2. BACKGROUND

In this section we first discuss the details of Tor's internal architecture, focusing on how data is sent through the network while covering some of the specifics of intra-relay communication. In addition, we review related work on improving Tor's performance through changes to the transport and scheduling mechanisms.

2.1 Tor's Architecture

The Tor overlay network consists of over 5,000 volunteer relays, providing anonymity by routing data through the network to the end destination, preventing any single intermediary from learning the identity of both the source and destination. Clients choose three relays - a guard, middle, and exit - and constructs *circuits* through them. TCP streams can then be multiplexed over the circuit to the exit relay that forwards the data on to the intended destination. Data transfered through the circuit is packaged into 512-byte cells and encrypted in a layered fashion, using shared symmetric keys with each of the three relays.

In order to create a circuit, the client sends a series of `EXTEND` cells through the circuit, each of which notifies the current last hop to extend the circuit to another relay. For example, the client sends an `EXTEND` cell to the guard telling it to extend to the middle. Afterwards the client sends another `EXTEND` to the middle telling it to extend the circuit to the exit. The relay, on receiving an `EXTEND` cell, will establish a *channel* to the next relay if one does not already exist. Cells from all circuits between the two relays get transfered over this channel, which is responsible for in-order delivery and, ideally, providing secure communication from potential eavesdroppers. Tor uses a TLS channel with a single TCP connection between the relays for in-order delivery and uses TLS to encrypt and authenticate all traffic.

Figure 1 shows the internal cell processing architecture of a Tor relay. Data is read from the channel and stored in an internal input buffer. Once enough data has been read (e.g. a full TLS record), the data is decrypted and the cells are sent to their respective circuit queues. Once a channel is able to write, it uses a prioritized circuit scheduler to select from all circuits travelling through the channel. Once a circuit is selected, it's queue is flushed to the output buffer on the channel. Once the output buffer has enough data it is then sent out over the channel to the next relay.

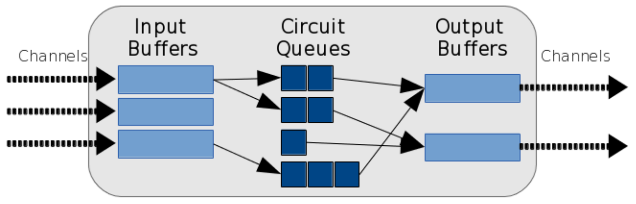


Figure 1: Internal architecture of cell processing inside a Tor relay. All cells from a given relay arrive on a single channel, and are then de-multiplexed to circuit queues for processing. Each circuit queue is emptied into the channel associated with the next hop relay

2.2 Related Work

The Tor network [7] has been very succesful in attracting users, servicing millions of clients daily. Since the network consists of volunteer relays with a very large user base, network resources such as bandwidth and processing are scarce. In order to sufficiently handle the increasing usage, a large body of research has been done looking into potential ways to lower latency and increase performance to the end users. These proposals range from dealing with the limitations of TCP [20, 18, 17], better utilization of network resources [4, 5], prioritizing web traffic over bulk data [3, 21, 14], improving ways of dealing with congestion [22, 4, 2], and adding incentives to act as a relay [12, 13]. We briefly cover the details of some of the research most relevant to the work in this paper.

TCP-over-DTLS: Reardon and Goldberg [20] were the first to measure and document several potential performance limitations due to the use of a single TCP connection for each channel, including head-of-line blocking, fate-sharing and buffering delay. In an attempt to circumvent these problems, they proposed using a Datagram TLS (DTLS) transport channel between relays to tunnel TCP connections through. In addition, each circuit gets its own (user-space) TCP connection instead of forcing all circuits to share a single (kernel) TCP connection. Reardon and Goldberg used micro-benchmarking to show that this design could significantly reduce the impact of these problems.

uTLS/uTCP: To address head-of-line blocking while avoiding problems associated with implementing a user-space TCP stack Nowlan *et al.*proposed to replace TLS channels with uTLS/uTCP [18, 19], a variant in the kernel implementation of TCP, allowing Tor to peek into the input buffer for full TLS records, decrypt them, and process them if the cell is the next one for some associated circuit. They evaluated this implementation using a "circuit-dumbell" topology, showing that head-of-line blocking was significantly reduced.

EWMA: Tang and Goldberg developed a prioritized circuit scheduler [21] using the exponential weighted moving average (EWMA) calculated on how often cells are sent on a circuit in an attempt to be able to prioritize interactive traffic over bulk traffic (e.g. BitTorrent). This is currently the scheduler used in the live Tor software. Jansen and Hopper [11] used the Shadow simulator to evaluate EWMA scheduling on a large-scale network and found that the benefits were highly dependent on the network load.

DiffTor: AlSabah, Bauer, and Goldberg [3] develop a novel approach in using classifiers to more accurately distinguish between web, bulk, and streaming traffic. Using their highly accurate classifiers, quality of service (QoS) algorithms can be used to give more opportune treatment to web traffic. The evaluation used throttling of bulk traffic to improve the throughput of other traffic classes, as suggested by Jansen *et al.* [14].

Traffic Splitting: AlSabah *et al.*propose Conflux, a dynamic loadbalancing algorithm [2] that can split traffic on one stream across multiple circuits in an attempt to avoid congestion in the Tor network. By embedding sequence numbers in the cell, the exit can reassemble the stream from the multiple circuits to provide in-order delivery to the end host.

Torchestra: To prevent bulk circuits from interfering with web circuits, Gopal and Heninger [8] developed a new channel, Torchestra, that creates two TLS connections between each relay, one reserved for web circuits and the other for bulk. This prevents head-of-line blocking that might be caused by bulk traffic from interfering with web traffic. The paper evaluates the Torchestra channel in a "circuit-dumbell" topology and shows that time to first byte and total download time for "interactive" streams decrease, while "heavy" streams do not see a significant change in performance.

PCTCP: Similar to TCP-over-DTLS, AlSabah and Goldberg [5] propose dedicating a separate TCP connection to each circuit and replacing the TLS session with an IPSEC tunnel that can then carry all the connections without letting an adversary learn circuit specific information from monitoring the different connections. This has the advantage of eliminating the reliance on user-space TCP stacks, leading to reduced implementation complexity and improved performance. However, as we show in the next section, the use of a kernel-provided socket for every circuit makes it possible to launch attacks that attempt to exhaust this resource at a targeted relay.

KIST: Jansen *et al.*[10] show that cells spend a large amount of time in the kernel output buffer, causing unneeded congestion and severely limiting the effect of prioritization in Tor. They introduce a new algorithm KIST with two main components: global scheduling across all writable circuits is done fixing circuit prioritization, and an autotuning algorithm that can dynamically determine how much data should be written to the kernel. This allows data to stay internal to Tor for longer, allowing it to make smarter scheduling decisions than simply dumping everything it can to the kernel, which operates in a FIFO manner.

3. SOCKET EXHAUSTION ATTACKS

This section discusses the extent to which Tor is vulnerable to socket descriptor exhaustion attacks that may lead to reductions in relay *availability* and client *anonymity*, explains how PCTCP creates a *new attack surface* with respect to socket exhaustion, and demonstrates how socket resource usage harms relay *stability*. The attacks in this section motivate the need for the intelligent management of sockets in Tor, which is the focus of Sections 4 and 5.

3.1 Sockets in Tor

On modern operating systems, file descriptors are a scarce resource that the kernel must manage and allocate diligently. On Linux, for example, soft and hard file limits are used to restrict the number of open file descriptors that any process may have open at one time. Once a process exceeds this limit, any system call that attempts to open a new file descriptor will fail and the kernel will return an `EMFILE` error code indicating too many open files. Since sockets are a specific type of file descriptor, this same issue can arise if a process opens sockets in excess of the file limit. Aware of this limitation, Tor internally utilizes its own connection limit. For relays running on Linux and BSD, an internal variable `ConnLimit` is set to the maximum limit as returned by the `getrlimit()` system call; the `ConnLimit` is set to a hard coded value of 15,000 on all other operating systems. Each time a socket is opened and closed, an internal counter is incremented and decremented; if a `tor_connect()` function call is made when this counter is above the `ConnLimit`, it preemptively returns an error rather than waiting for one from the `connect` system call.

3.2 Attack Strategies

There are several cases to consider in order to exploit open sockets as a relay attack vector. Relay operators may be: (i) running Linux with the default maximum descriptor limit of 4096; (ii) running Linux with a custom descriptor limit or running a non-Linux OS with the hard-coded `ConnLimit` of 15,000; and (iii) running any OS and allowing unlimited descriptors. We note that setting a custom limit generally requires root privileges, although it does not require that Tor itself be run as the root user. Also note that each Tor relay connects to every other relay with which it communicates, leading to potentially thousands of open sockets under normal operation. In any case, the adversary's primary goal is to cause a victim relay to open as many sockets as possible.

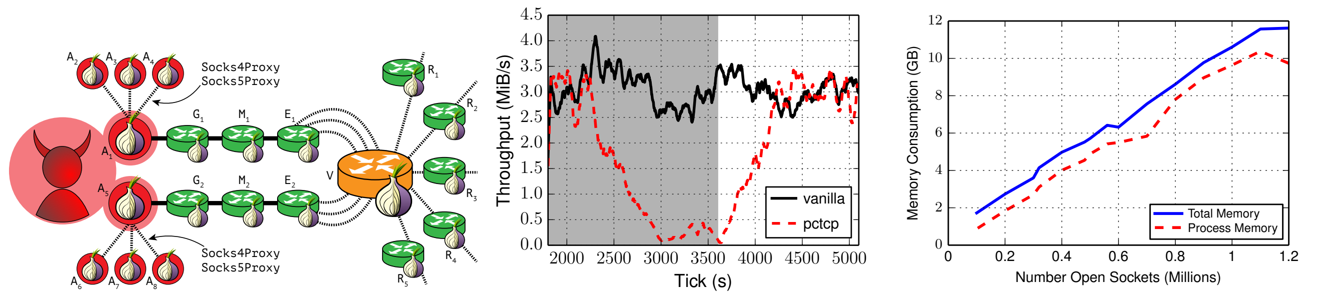
Consuming Sockets at Exit Relays: In order to consume sockets at an exit relay, an adversary can create multiple circuits through independent paths and request TCP streams to various destinations. Ideally, the adversary would select services that use persistent connections to ensure that the exit holds open the sockets. The adversary could then send the minimal amount required to keep the connections active. Although the adversary remains anonymous (because the victim exit relay does not learn the adversary's identity), keeping persistent connections active so that they are not closed by the exit will come at a bandwidth cost.

Consuming Sockets at Any Relay: Bandwidth may be traded for CPU and memory by using Tor itself to create the persistent connections, in which case relays in any position may be targeted. This could be achieved by an adversary connecting several Tor client instances directly to a victim relay; each such connection would consume a socket descriptor. However, the victim would be able to determine the adversary's IP address (i.e., identity). The attack can also be done anonymously. The basic mechanism to do so was outlined in The Sniper Attack, Section II-C-3 [15], where it was used in a relay memory consumption denial of service attack. Here, we use similar techniques to anonymously consume sockets descriptors at the victim.

The attack is depicted in Figure 2a. First, the adversary launches several Tor client sybils. A_1 and A_5 are used to build independent circuits through G_1, M_1, E_1 and G_2, M_2, E_2, respectively, following normal path selection policies. These sybil clients also configure a `SocksPort` to allow connections from other applications. Then, A_2, A_3, and A_4 use either the `Socks4Proxy` or `Socks5Proxy` options to extend new circuits to a victim V through the Tor circuit built by A_1. The A_6, A_7, and A_8 sybils similarly extend circuits to V through the circuit built by A_5. Each Tor sybil client will create a new tunneled channel to V, causing the exits E_1 and E_2 to establish new TCP connections with V.

Each new TCP connection to V will consume a socket descriptor at the victim relay. When using either the `Socks4Proxy` or the `Socks5Proxy` options, the Tor software manual states that "Tor will make all OR connections through the SOCKS [4,5] proxy at host:port (or host:1080 if port is not specified)." We also successfully verified this behavior using Shadow. This attack allows an adversary to consume roughly one socket for every sybil client, while remaining anonymous from the perspective of the victim. Further, the exits E_1 and E_2 will be blamed if any misbehavior is suspected, who themselves will be unable to discover the identity of the true attacker. If Tor were to use a new socket for every circuit, as suggested by PCTCP [5], then the adversary could effectively launch a similar attack with only a single Tor client.

Consuming Sockets with PCTCP: PCTCP may potentially offer performance gains by dedicating a separate TCP connection for every circuit. However, PCTCP widens the attack surface and reduces the cost of the anonymous socket exhaustion attack dis-



(a) **(b)** **(c)**

Figure 2: Showing **(a)** anonymous socket exhaustion attack using client sybils **(b)** throughput of relay when launching a socket exhaustion attack via circuit creation with shaded region representing when the attack was being launched **(c)** memory consumption as a process opens sockets using libevent.

cussed above. If all relays use PCTCP, an adversary may simply send `EXTEND` cells to a victim relay through any other relay in the network, causing a new circuit to be built and therefore a socket descriptor opened at the victim. Since the cells are being forwarded from other relays in the network, the victim relay will not be able to determine who is originating the attack. Further, the adversary gets long-term persistent connections cheaply with the use of the Tor config options `MaxClientCircuitsPending` and `CircuitIdleTimeout`. The complexity of the socket exhaustion attack is reduced and the adversary no longer needs to launch the tunneled sybil attack in order to anonymously consume the victim's sockets. By opening circuits with a single client, the attacker will cause the victim's number of open connections to reach the `ConnLimit` or may cause relay stability problems (or both).

3.3 Effects of Socket Exhaustion

Socket exhaustion attacks may lead to either reduced relay availability and client anonymity if there is a descriptor limit in place, or may harm relay stability if either there is no limit or the limit is too high. We now explore these effects.

Limited Sockets: If there is a limit in place, then opening sockets will consume the shared descriptor resource. An adversary that can consume all sockets on a relay will have effectively made that relay unresponsive to connections by other honest nodes due to Tor's `ConnLimit` mechanism. If the adversary can persistently maintain this state over time, then it has effectively disabled the relay by preventing it from making new connections to other Tor nodes.

We ran a socket consumption attack against both vanilla Tor and PCTCP using Shadow and the network model described in Section 5.1. Our attacker node created 1000 circuits every 6 seconds through a victim relay, starting at time 1800 and ending at time 3600. Figure 2b shows the victim relay's throughput over time as new circuits are built and victim sockets are consumed. After consuming all available sockets, the victim relay's throughput drops close to 0 as old circuits are destroyed, effectively disabling the relay. This, in turn, will move honest clients' traffic away from the relay and onto other relays in the network. If the adversary is running relays, then it has increased the probability that its relays will be chosen by clients and therefore has improved its ability to perform end-to-end traffic correlation [16]. After the attacker stops the attack at time 3600, the victim relay's throughput recovers as clients' are again able to successfully create circuits through it.

Unlimited Sockets: One potential solution to the availability and anonymity concerns caused by a file descriptor limit is to remove the limit (i.e., set an unlimited limit), meaning that `ConnLimit` gets set to 2^{32} or 2^{64}. At the lesser of the two values, and assuming that an adversary can consume one socket for every 512-byte Tor cell it sends to a victim, it would take around 2 TiB of network bandwidth to cause the victim to reach its `ConnLimit`. However, even if the adversary cannot cause the relay to reach its `ConnLimit`, opening and maintaining sockets will still drain the relay's physical resources, and will increase the processing time associated with socket-based operations in the kernel. By removing the open descriptor limit, a relay becomes vulnerable to performance degradation, increased memory consumption, and an increased risk of being killed by the kernel or otherwise crashing. An adversary may cause these effects through the same attacks it uses against a relay with a default or custom descriptor limit.

Figure 2c shows how memory consumption increases with the number of open sockets as a process opens over a million sockets. We demonstrate other performance effects using a private Tor network of 5 machines in our lab. Our network consisted of 4 relays total (one directory authority), each running on a different machine. We configured each relay to run Tor `v0.2.5.2-alpha` modified to use a simplified version of PCTCP that creates a new OR connection for every new circuit. We then launched a simple file server, a Tor client, and 5 file clients on the same machine as the directory authority. The file clients download arbitrary data from the server through a specified path of the non-directory relays, always using the same relay in each of the entry, middle, and exit positions. The final machine ran our Tor attacker client that we configured to accept localhost connections over the `ControlPort`. We then used a custom python script to repeatedly: (1) request that 1000 new circuits be created by the Tor client, and (2) pause for 6 seconds. Each relay tracked socket and bandwidth statistics; we use throughput and the time to open new sockets to measure performance degradation effects and relay instability.

The stability effects for the middle relay are shown in Figure 3. The attack ran for just over 2500 seconds and caused the middle relay to successfully open more than 50 thousand sockets. We noticed that our relays were unable to create more sockets due to port allocation problems, meaning that (1) we were unable to measure the potentially more serious performance degradation effects that occur when the socket count exceeds 65 thousand, and (2) unlimited sockets may not be practically attainable due to port exhaustion between a pair of relays. Figure 3a shows throughput over time and Figure 3b shows a negative correlation of bandwidth to the number of open sockets; both of these figures show a drop of more than 750 KiB/s in the 60 second moving average throughput during our experiment. Processing overhead during `socket` system calls over

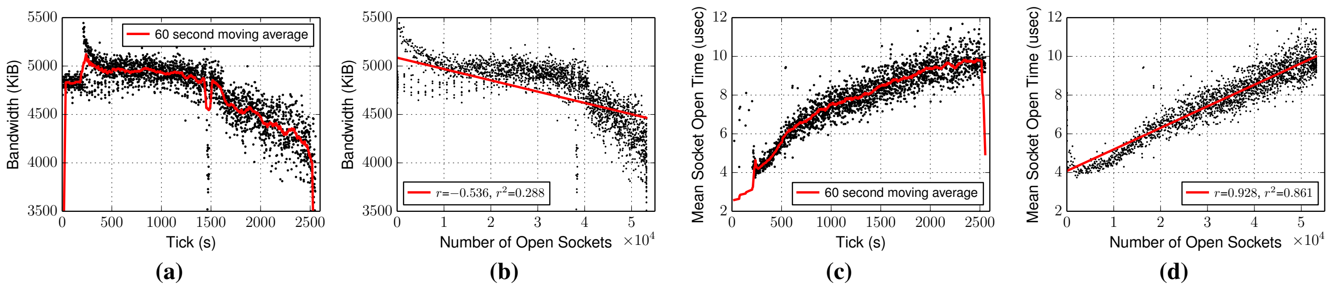


Figure 3: Showing **(a)** throughput over time **(b)** linear regression correlating throughput to the number of open sockets **(c)** kernel time to open new sockets over time **(d)** linear regression correlating kernel time to open a new socket to the number of open sockets.

time is shown in Figure 3c, and the correlation to the number of open sockets is shown in Figure 3d; both of these figures clearly indicate that increases in kernel processing time can be expected as the number of open sockets increases. Although the absolute time to open sockets is relatively small, it more than doubled during our experiment; we believe this is a strong indication of performance degradation in the kernel and that increased processing delays in other kernel socket processing functions are likely as well.

4. IMUX

This section explores a new algorithm that takes advantage of multiple connections while respecting the `ConnLimit` imposed by Tor and preventing the attacks discussed above in Section 3. Both Torchestra and PCTCP can be seen as heuristically derived instances of a more general resource allocation scheme with two components, one determining how many connections to open between relays, and the second in selecting a connection to schedule cells on. Torchestra's heuristic is to fix the number of connections at two, designating one for light traffic and the other for heavy, then scheduling cells based on the traffic classification of each circuit. PCTCP keeps a connection open for each circuit between two relays, with each connection devoted to a single circuit that schedules cells on it.

While there is a demonstrable advantage to being able to open multiple connections between two communicating relays, it is important to have an upper limit on the number of connections allowed to prevent anonymous socket exhaustion attacks against relays, as shown in Section 3. In this section we introduce IMUX, a new heuristic for handling multiple connections between relays that is able to dynamically manage open connections while taking into consideration the internal connection limit in Tor.

4.1 Connection Management

Similar to PCTCP, we want to ensure the *allocation* of connections each channel has is proportional to the number of active circuits each channel is carrying, dynamically adjusting as circuits are opened and closed across all channels on the relay. PCTCP can easily accomplish this by dedicating a connection to each circuit every time one is opened or closed, but since IMUX enforces an upper limit on connections, the connection management requires more care, especially since both ends of the channel will may have different upper limits.

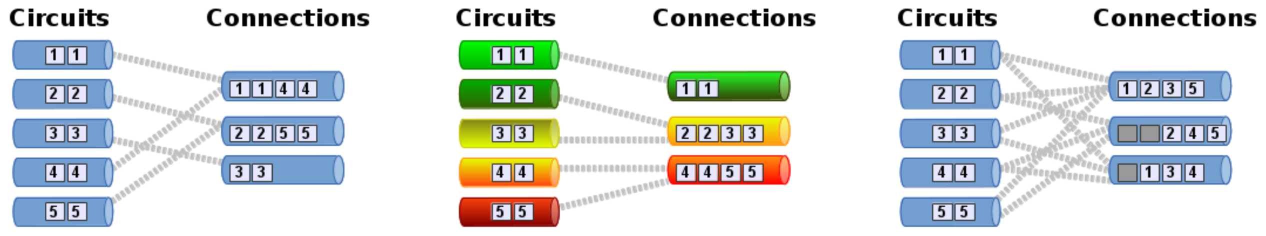
We first need a protocol dictating how and when relays can open and close connections. During the entire time the channel is open, only one relay is allowed to open connections, initially set to the relay that creates the channel. However, at any time either relay may close a connection if it detects the number of open sockets approaching the total connection limit. When a relay decides to

Algorithm 1 Function to determine the maximum number of connections that can be open on a channel.

```
 1: function GETMAXCONNS($nconns$, $ncircs$)
 2:     $totalCircs \leftarrow len(globalActiveCircList)$
 3:     if $ncircs$ is 0 or $totalCircs$ is 0 then
 4:         return 1
 5:     end if
 6:     $frac \leftarrow ncircs/totalCircs$
 7:     $totalMaxConns \leftarrow ConnLimit \cdot \tau$
 8:     $connsLeft \leftarrow totalMaxConns - n\_open\_sockets()$
 9:     $maxconns \leftarrow frac \cdot totalMaxConns$
10:     $maxconns \leftarrow MIN(maxconns, nconns \cdot 2)$
11:     $maxconns \leftarrow MIN(maxconns,$
                                  $nconns + connsLeft)$
12:     return $maxconns$
13: end function
```

close a connection, it must first decide *which* connection should be closed. To pick which connection to close the algorithm uses three criteria for prioritizing available connections: (1) Always pick connections that haven't fully opened yet; (2) Among connections with state `OPENING`, pick the one that was created most recently; and (3) If all connections are open, pick the one that was least recently used. In order to close a connection C, first the relay must make sure the relay on the other end of the channel is aware C is being closed to prevent data from being written to C during the closing process. Once a connection C is chosen, the initiating relay sends out an empty cell with a new command, `CLOSING_CONN_BEGIN`, and marks C for close to prevent any more data from being written to it. Once the responding relay on the other end of the channel receives the cell, it flushes any data remaining in the buffer for C, sends back another `CLOSING_CONN_END` cell to the initiating relay, and closes the connection. Once the initiating relay receives the `CLOSING_CONN_END`, it knows that it has received all data and is then able to proceed closing the socket.

Once a channel has been established, a housekeeping function is called on the channel every second that then determines whether to open or close any connections. The function to determine the maximum number of connections that can open on a channel can be seen in Algorithm 1. We calculate a soft upper limit on the total number of allowed open connections on the relay by taking `ConnLimit` and multiplying it by the parameter $\tau \in (0, 1)$. `ConnLimit` is an internal variable that determines the maximum number of sockets allowed to be open on the relay. On Linux based relays this is set by calling `getrlimit()` to get the file limit on the machine, otherwise it is fixed at 15,000. The parameter τ is a threshold value between 0 and 1 that sets a soft upper limit on the number of open connections. Since once the number of open connections exceeds



(**a**) Circuit Round Robin Scheduler (**b**) EWMA Mapping Scheduler (**c**) Shortest Queue Scheduler

Figure 4: The three different connection scheduling algorithms used in IMUX.

ConnLimit all connect() calls will fail, we want some breathing room so new channels and connections can still be opened, temporarily going past the soft limit, until other connections can be closed to bring the relay back under the limit. To calculate the limit for the channel we simply take this soft limit on the total of all open connections and multiply it by the fraction of active circuits using the channel. This gives us an upper bound on the connection limit for the channel. We then take the minimum of this upper limit with the number of current open connections on the channel multiplied by two. This is done to prevent rapid connection opening when a channel is first created, particularly when the other relay has a much lower connection limit. Finally we take the minimum of that calculation with the number of open connections plus the number of connections that can be opened on the relay before hitting the soft limit (that could be negative, signaling that connections need to be *closed*). Otherwise the channel could create too many connections, driving the number of open connections past ConnLimit.

The housekeeping function is called every second and determines if any connections need to be opened or closed. If we have two few connections and, based on the protocol discussed in the previous paragraph, the relay is allowed to open connections on this channel, enough connections are created to match the maximum connections allowed. If we have too *many* connections open, we simply close enough until we are at the connection limit. We use the previously discussed protocol for selecting and closing connections, prioritizing newly created connections in an attempt to close unneeded connections before the TLS handshake is started, preventing unnecessary overhead. In addition to the housekeeping function, whenever a channel accepts an incoming connection, it also checks to see if the number of connections exceeds the maximum allowed returned by Algorithm 1. If so, it simply notifies the relay at the other end of the channel that it is closing the connection, preventing that relay from opening any more connections as dictated by the protocol.

4.2 Connection Scheduler

For connection scheduling, PCTCP assigns each circuit a dedicated connection to schedule cells on. Torchestra schedules cells from a circuit to either the light or heavy connection depending on how much data is being sent through the circuit. A circuit starts out on the light connection, and if at some point its EWMA value crosses a threshold it is switched to the heavy connection. To accommodate this, a switching protocol is introduced so the relay downstream can be notified when a circuit has switched and on what connection it can expect to receive cells.

While scheduling cells from a circuit on a single connection makes in-order delivery easier by relying on TCP, with multiple connections per channel it is not necessary to do so and in fact may not be optimal to keep this restraint. Similar to the uTLS implementation in Tor [19] and Conflux [2], we embed an 8-byte sequence number in the relay header of all cells. This allows the channel to schedule cells across multiple connections that then get reordered at the other end of the channel. With sequence numbers in place and the capability to schedule cells from a circuit across multiple connections, we can evaluate different scheduling algorithms attempting to increase throughput or "fairness", for example, where low traffic circuits have better performance than high traffic ones. We will briefly cover the different algorithms and heuristics we can use.

Circuit Round Robin: The first scheduler, shown in Figure 4a, emulates PCTCP by assigning each circuit a single connection to transmit its cells. When circuits are added to a channel, the scheduler iterates round robin style through the circuit list, assigning circuits to successive connections. If there are more circuits than connections some circuits will share a single connection. When a connection is closed, any circuit assigned to it will be given a new one to schedule through, with the connections remaining iterated in the same round robin style as before.

EWMA Mapping: Internal to Tor is a circuit scheduling algorithm proposed by Tang and Goldberg [21] that uses an exponential moving weight average (EWMA) algorithm to compute how "noisy" circuits are being, and then schedule them from quietest to loudest when choosing what circuits to flush. Using the same algorithm, we compute the EWMA value for each connection as well as the circuits. Then, as seen in Figure 4b, the circuits and connections are ordered from lowest to highest EWMA value and we attempt to map the circuits to a connection with a similar EWMA value. More specifically, after sorting the circuits, we take the rank of the circuit $1 \leq r_i \leq ncircs$ and compute the percentile $p_i = \frac{r_i}{ncircs}$. We do the same thing with the connections computing their percentiles denoted C_j. Then to determine which connection to map a circuit to, we pick the connection j such that $C_{j-1} < p_i \leq C_j$.

Shortest Queue: While the EWMA mapping scheduler is built around the idea of "fairness", where we penalize high usage circuits by scheduling them on busier connections, we can construct an algorithm aimed at increasing overall throughput by always scheduling cells in an opportunistic manner. The shortest queue scheduler, shown in Figure 4c, calculates the queue length of each connection and schedules cells on connections with the shortest queue. This is done by taking the length of the internal output buffer queue that Tor keeps for each connections, and adding it with the kernel TCP buffer that each socket has; this is obtained using the ioctl[1] function call and passing it the socket descriptor and TIOCOUTQ.

[1]http://man7.org/linux/man-pages/man2/ioctl.2.html

4.3 KIST: Kernel-Informed Socket Transport

Recent research by Jansen *et al.*[10] showed that by minimizing the amount of data that gets buffered inside the kernel and instead keeping it local to Tor, better scheduling decisions can be made and connections can be written in an opportunistic manner increasing performance. The two main components of the algorithm are global scheduling and autotuning. In vanilla Tor, libevent iterates through the connections in a round robin fashion, notifying Tor that the connection can write. When Tor receives this notification, it performs circuit scheduling *only* on circuits associated with the connection. Global scheduling takes a list of all connections that can write, and schedules between circuits associated with every single connection, making circuit prioritization more effective. Once a circuit is chosen to write to a connection, autotuning then determines how much data should be flushed to the output buffer and onto the kernel. By using a variety of socket and TCP statistics, it attempts to write just enough to keep data in the socket buffer at all times, without flushing everything, giving more control to Tor for scheduling decisions.

KIST can be used in parallel with Torchestra, PCTCP, and IMUX, and also as a connection manager within the IMUX algorithm. After KIST selects a circuit and decides how much data to flush, Torchestra and PCTCP make their own determination on which connection to write to based on their internal heuristics. IMUX can also take into account the connection selected by the KIST algorithm and use that to schedule cells from the circuit. Similar to the other connection schedulers, this means that circuits can be scheduled across different connections in a more opportunistic fashion.

5. EVALUATION

In this section we discuss our experimental setup, the details of our implementations of Torchestra and PCTCP (for comparison with IMUX), evaluate how the dynamic connection management in IMUX is able to protect against potential denial of service attacks by limiting the number of open connections, and finally compare performance across the multiple connection schedulers, along with both Torchestra and PCTCP.

5.1 Experimental Setup

We perform experiments in Shadow [1, 11], a discrete event network simulator capable of running real Tor code in a simulated network. Shadow allows us to create large-scale network deployments that can be run locally and privately on a single machine, avoiding privacy risks associated with running on the public network that many active users rely on for anonymity. Because Shadow runs the Tor software, we are able to implement our performance enhancements as patches to Tor.[2] We also expect that Tor running in Shadow will exhibit realistic application-level performance effects including those studied in this paper. Finally, Shadow is deterministic; therefore our results may be independently reproduced by other researchers. Shadow also enables us to isolate performance effects and attribute them to a specific set of configurations, such as variations in scheduling algorithms or parameters. This isolation means that our performance comparisons are meaningful independent of our ability to precisely model the complex behaviors of the public Tor network.

We initialize a Tor network topology and node configuration and use it as a common Shadow deployment base for all experiments in this section. For this common base, we use the *large* Tor configuration that is distributed with Shadow.[3] The techniques for producing this model are discussed in detail in [9] and updated in [10]. It consists of 500 relays, 1350 web clients, 150 bulk clients, 300 perf clients and 500 file servers. Web clients repeatedly download a 320 KiB file while pausing between 1 to 60 seconds after every download. Bulk clients continuously download 5 MiB files with no pausing between downloads. The perf clients download a file every 60 seconds, with 100 downloading a 50 KiB file, 100 downloading a 1 MiB file, and 100 download a 5 MiB file.

The Shadow perf clients are configured to mimic the behavior of the TorPerf clients that run in the public Tor network to measure Tor performance over time. Since the Shadow and Tor perf clients download files of the same size, we verified that the performance characteristics in our Shadow model were reasonably similar to the public network.

5.2 Implementations

In the original Torchestra design discussed in [8], the algorithm uses EWMA in an attempt to classify each circuit as “light” or “heavy”. Since the EWMA value will depend on many external network factors (available bandwidth, network load, congestion, etc.), the algorithm uses the average EWMA value for the light and heavy connection as benchmarks. Using separate threshold values for the light and heavy connection, when a circuit either goes above or below the average multiplied by the threshold the circuit is reclassified and is swapped to the other connection. The issue with this, as noted in [3], is that web traffic tends to be bursty causing temporary spikes in circuit EWMA values. When this occurs it increases the circuit’s chance of becoming misclassified and assigned to the bulk connection. Doing so will in turn *decrease* the average EWMA of both the light and bulk connections, making it *easier* for circuits to exceed the light connections threshold and *harder* for circuits to drop below the heavy connection threshold, meaning web circuits that get misclassified temporary will find it more difficult to get reassigned to the light connection. A better approach would be to use a more complex classifier such as DiffTor [3] to determine if a circuit was carrying web or bulk traffic. For our implementation, we have Torchestra use an idealized version of DiffTor where relays have perfect information about circuit classification. When a circuit is first created by a client, the client sends either a `CELL_TRAFFIC_WEB` or `CELL_TRAFFIC_BULK` cell notifying each relay of the type of traffic that will be sent through the circuit. Obviously this would be unrealistic to have in the live Tor network, but it lets us examine Torchestra under an ideal situation.

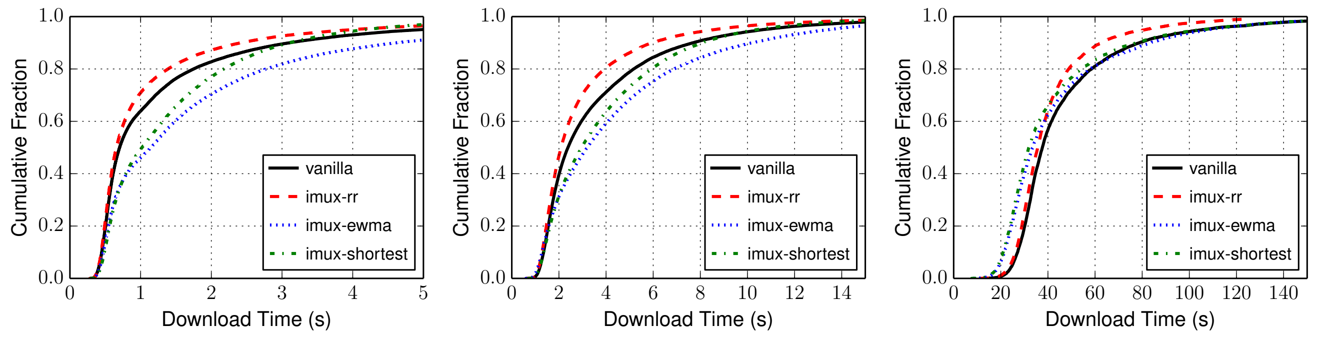
For PCTCP there are two main components of the algorithm. First is the dedicated connection that gets assigned to each circuit, and the second is replacing per connection TLS encryption with a single IPSec layer between the relays, preventing an attacker from monitoring a single TCP connection to learn information about a circuit. For our purposes we are interested in the first component, performance gains from dedicating a connection to each circuit. The IPSec has some potential to increase performance, since each connection no longer requires a TLS handshake that adds some overhead, but there are a few obstacles noted in [5] that could hinder deployment. Furthermore, it can be deployed alongside any algorithm looking to open multiple connections between relays. For simplicity, our PCTCP implementation simply opens a new TLS connection for each circuit created that will use the new connection exclusively for transferring cells.

5.3 Connection Management

One of the main goals of the dynamic connection manager in IMUX is to avoid denial of service attacks by consuming all available open sockets. To achieve this IMUX has a soft limit that caps

[2]We modified Tor `v0.2.5.2-alpha`.

[3]We use Shadow `v1.9.2`, the latest release as of this writing.



(a) Time to first byte **(b)** Time to last byte of 320KiB **(c)** Time to last byte of 5MiB

Figure 5: Performance comparison of IMUX connection schedulers

the total number of connections at `ConnLimit` $\cdot\tau$, where τ is a parameter between 0 and 1. If this is set too low, we may lose out in potential performance gains that go unrealized, while if it is too high we risk exceeding the hard limit `ConnLimit`, causing new connections to error out leaving ourselves open to denial of service attacks. During our experiments we empirically observed that $\tau = 0.9$ was the highest the parameter could be set without risking crossing `ConnLimit`, particularly when circuits were being created rapidly causing high connection churn.

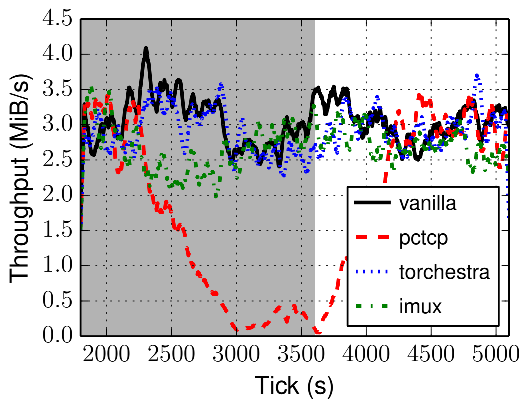


Figure 6: Comparing socket exhaustion attack in vanilla Tor with Torchestra, PCTCP and IMUX.

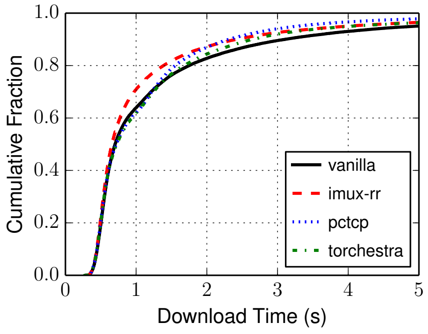
Figure 6 shows the effects of the socket exhaustion attack discussed in Section 3 with Torchestra and IMUX included. Since Torchestra simply opens two connections between each relay, the attack is not able to consume all available sockets, leaving the relay unaffected. The IMUX results show an initial slight drop in throughput as many connections are being created and destroyed. However, throughput recovers to the levels achieved with vanilla Tor and Torchestra as the connection manager stabilizes.

5.4 Performance

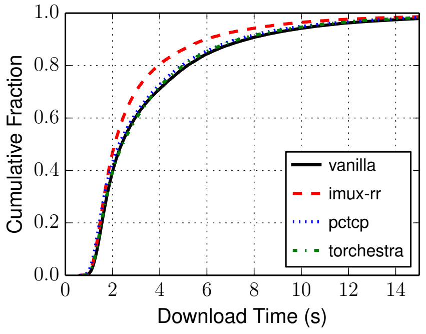
First we wanted to determine how the various connection schedulers covered in Section 4.2 perform compared to each other and against vanilla Tor. Figure 5 shows the results of large scale experiments run with each of the connection schedulers operating with IMUX. Round robin clean performs better than both EWMA mapping and the shortest queue connection schedulers, at least with respect to time to first byte and download time for web clients shown in Figures 5a and 5b. This isn't completely surprising for the shortest queue scheduler as it indiscriminately tries to push as much data as possible, favoring higher bandwidth traffic at the potential cost of web traffic performance. The EWMA mapping scheduler produces slightly better results for web traffic compared to shortest queue scheduling, but it still ends up performing worse than vanilla Tor. This is related to the issue with using EWMA in Torchestra for classification, that web traffic tends to be sent in large bursts causing the EWMA value to spike rapidly that then decreases over time. So while under the EWMA mapping scheme the first data to be sent will be given high prioritization, as the EWMA value climbs the data gets sent to busier connections causing the total time to last byte to decrease as a consequence.

Figure 7 shows the download times when using IMUX with round robin connection scheduling against vanilla Tor, Torchestra and PCTCP. While Torchestra and PCTCP actually perform identically to vanilla Tor, IMUX sees an increase in performance both for web and bulk downloads. Half of the web clients see an improvement of at least 12% in their download times, with 29% experiencing more than 20% improvement, with the biggest reduction in download time seen at the 75^{th} percentile, dropping from 4.5 seconds to 3.3 seconds. Gains for bulk clients are seen too, although not as large; around 10% of clients seeing improvements of 10-12%. Time to first byte across all clients improves slightly as shown in Figure 7a, with 26% of clients seeing reductions ranging from 20-23% when compared to vanilla Tor, that then drops down to 11% of clients who see the same level of improvements compared to Torchestra and PCTCP.

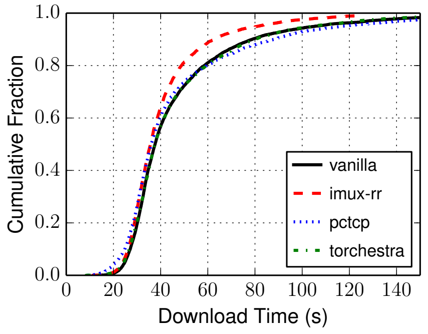
We then ran large experiments with KIST enabled, with IMUX using KIST for connection scheduling. While overall download times improved from the previous experiments, IMUX saw slower download times for web clients and faster downloads for bulk clients, as seen in Figure 8. 40% of clients see an *increase* in time to first byte, while 87% of bulk client see their download times decrease from 10-28%. This is due to the fact that one of the main advantages to using multiple connections per channel is that it prevents bulk circuits from forcing web circuits to hold off on sending data due to packet loss the bulk circuit caused. In PCTCP, for example, this will merely cause the bulk circuits connection to become throttled while still allowing all web circuits to send data. Since KIST forces Tor to hold on to cells for longer and only writes a minimal amount to the kernel, it's able to make better scheduling decisions, preventing web traffic from unnecessarily buffering behind bulk traffic. Furthermore, KIST is able to take packet loss into consideration since it uses the TCP congestion window in calculat-



(a) Time to first byte

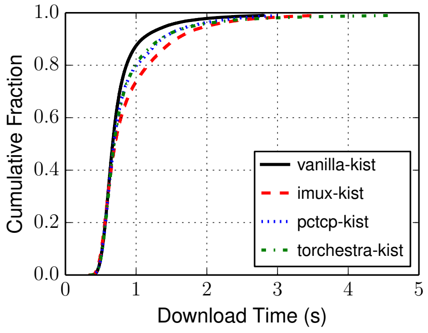


(b) Time to last byte of 320KiB

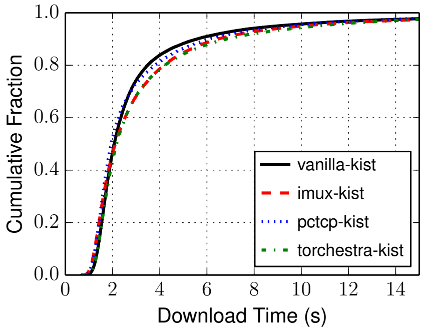


(c) Time to last byte of 5MiB

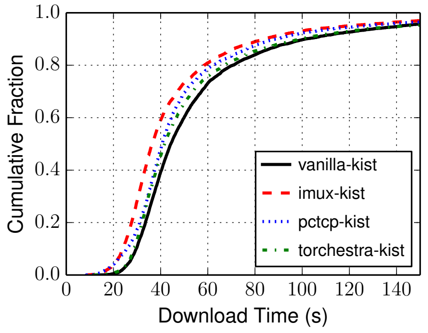
Figure 7: Performance comparison of IMUX to Torchestra [8] and PCTCP [5]



(a) Time to first byte



(b) Time to last byte of 320KiB



(c) Time to last byte of 5MiB

Figure 8: Performance comparison of IMUX to Torchestra [8] and PCTCP [5], all using the KIST performance enhancements [10]

ing how much to write to the socket. Since the congestion window is reduced when there is packet loss, KIST will end up writing a smaller amount of data whenever packet loss occurs.

6. DISCUSSION

In this section we discuss the potential for an adversary to game the IMUX algorithm, along with the limitations on the protections against the denial of service attacks discussed in Section 3 and some possible ways to protect against them.

Active Circuits: The IMUX connection manager distributes connections to channels based on the fraction of *active* circuits contained on each channel. An adversary could game the algorithm by artificially increasing the number of active circuits on a channel they're using, heavily shifting the distribution of connections to the channel and increase throughput. The ease of the attack depends heavily on how we define an "active circuit", we can do one of three ways: (1) the number of open circuits that haven't been destroyed; (2) the number of circuits that has sent a minimal number of cells, measured in raw numbers of using an EWMA with a threshold; or (3) using DiffTor we can only consider circuits classified as web or bulk. No matter what definition is used an adversary will still technically be able to game the algorithm, the major difference is the amount of bandwidth that needs to be expended by the adversary to accomplish their task. If we just count the number of open circuits, an adversary could very easily restrict all other channels to only one connection, while the rest are dedicated to the channel they're using. Using an EWMA threshold or DiffTor classifier requires the adversary to actually send data over the circuits, with the amount determined by what thresholds are in place. So while the potential to game the algorithm will always exist, the worse an adversary can do is reduce all other IMUX channels to one connection, the same as how vanilla Tor operates.

Defense Limitations: By taking into account the connection limit of the relay, the dynamic connection manager in IMUX is able to balance the performance gains realized by opening multiple connections while protecting against the new attack surface made available with PCTCP that lead to a low-bandwidth denial of service attack against any relay in the network. However there still exists potential socket exhaustion attacks inherent to how Tor operates. The simplest of these simply requires opening streams through a targeted exit, causing sockets to be opened to any chosen destination. Since this is a fundamental part of how an exit relay operates, there is little that can be done to directly defend against this attack, although it can be made potentially more difficult to perform. Exit relays can attempt keep sockets short lived and close ones that have been idle for a short period of time, particularly when close to the connection limit. They can also attempt to prioritize connections between relays instead of ones exiting to external servers. While preventing access at all is undesirable, this may be the lesser of two evils, as it will still allow the relay to participate in the Tor network, possibly preventing adversarial relays from being chosen. This kind of attack only affects exit relays, however the technique utilizing the `Socks5Proxy` option can target any relay in the network. Since this is performed by tunneling OR connections through a circuit, the attack is in effect anonymous, meaning relays cannot simply utilize IP blocking to protect against it. One potential solution is to require clients to solve computationally intense puzzles in order to create a circuit as proposed by Barbera *et al.*[6]. This reduces the ease that a single adversary is able to mass produce circuits, resulting in socket descriptor consumption. Additionally, since this attack requires the client to send EXTEND cells through the exit to initiate a connection through the targeted relay, exits could simply disallow connections back into the Tor network for circuit creation. This would force an adversary to have to directly connect to whichever non-exit relay they were targeting, in which case IP blocking becomes a viable strategy to protect against such an attack once it is detected.

7. CONCLUSION

In this paper we present a new class of socket exhaustion attacks that allow an adversary to anonymously perform a denial of service attacks against relays in the Tor network. We outline how PCTCP, a new transport proposal, introduces a new attack surface in this new class of attacks. In response, we introduce a new protocol, IMUX, generalizing the designs of PCTCP and Torchestra, that is able to take advantage of opening multiple connections between relays while still able to defend against these socket exhaustion attacks. Through large scale experiments we evaluate a series of connection schedulers operating within IMUX, look at the performance of IMUX with respect to vanilla Tor, Torchestra and PCTCP, and investigate how all these algorithms operate with a newly proposed prototype, KIST.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. We are very grateful to Max Schuchard and Shuai Li with assistance in designing and implementing the memory consumption experiments, and Leiah Jansen for creating the depiction of the anonymous socket exhaustion attack. This work was partially supported by NSF through grant CNS-1314637.

8. REFERENCES

[1] Shadow Homepage and Code Repositories. https://shadow.github.io/, https://github.com/shadow/.

[2] ALSABAH, M., BAUER, K., ELAHI, T., AND GOLDBERG, I. The path less travelled: Overcoming tor's bottlenecks with traffic splitting. In *Proceedings of the 13th Privacy Enhancing Technologies Symposium (PETS 2013)* (July 2013).

[3] ALSABAH, M., BAUER, K., AND GOLDBERG, I. Enhancing tor's performance using real-time traffic classification. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)* (October 2012).

[4] ALSABAH, M., BAUER, K., GOLDBERG, I., GRUNWALD, D., MCCOY, D., SAVAGE, S., AND VOELKER, G. Defenestrator: Throwing out windows in tor. In *Proceedings of the 11th Privacy Enhancing Technologies Symposium (PETS 2011)* (July 2011).

[5] ALSABAH, M., AND GOLDBERG, I. PCTCP: Per-Circuit TCP-over-IPsec Transport for Anonymous Communication Overlay Networks. In *Proceedings of the 20th ACM conference on Computer and Communications Security (CCS 2013)* (November 2013).

[6] BARBERA, M. V., KEMERLIS, V. P., PAPPAS, V., AND KEROMYTIS, A. D. Cellflood: Attacking tor onion routers on the cheap. In *Computer Security–ESORICS 2013*. Springer, 2013, pp. 664–681.

[7] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium* (August 2004).

[8] GOPAL, D., AND HENINGER, N. Torchestra: Reducing interactive traffic delays over tor. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2012)* (October 2012), ACM.

[9] JANSEN, R., BAUER, K., HOPPER, N., AND DINGLEDINE, R. Methodically modeling the tor network. In *Proceedings of the USENIX Workshop on Cyber Security Experimentation and Test (CSET 2012)* (August 2012).

[10] JANSEN, R., GEDDES, J., WACEK, C., SHERR, M., AND SYVERSON, P. Never been KIST: Tor's congestion management blossoms with kernel-informed socket transport. In *Proceedings of the 23rd conference on USENIX security symposium* (2014), USENIX Association.

[11] JANSEN, R., AND HOPPER, N. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Proc. of the 19th Network and Distributed System Security Symposium* (2012).

[12] JANSEN, R., HOPPER, N., AND KIM, Y. Recruiting new Tor relays with BRAIDS. In *Proceedings of the 2010 ACM Conference on Computer and Communications Security (CCS 2010)* (October 2010), A. D. Keromytis and V. Shmatikov, Eds., ACM.

[13] JANSEN, R., JOHNSON, A., AND SYVERSON, P. LIRA: Lightweight incentivized routing for anonymity. In *20th Annual Network & Distributed System Security Symposium (NDSS '13)* (2013), Internet Society.

[14] JANSEN, R., SYVERSON, P., AND HOPPER, N. Throttling tor bandwidth parasites. In *Proceedings of the 21st USENIX Security Symposium* (August 2012), USENIX Association.

[15] JANSEN, R., TSCHORSCH, F., JOHNSON, A., AND SCHEUERMANN, B. The sniper attack: Anonymously deanonymizing and disabling the Tor network. In *Proceedings of the Network and Distributed Security Symposium - NDSS '14* (February 2014), IEEE.

[16] JOHNSON, A., WACEK, C., JANSEN, R., SHERR, M., AND SYVERSON, P. Users get routed: Traffic correlation on tor by realistic adversaries. In *Proceedings of the 20th ACM conference on Computer and Communications Security (CCS 2013)* (November 2013).

[17] MURDOCH, S. J. Comparison of Tor datagram designs. Tech Report 2011-11-001, Tor, November 2011. http://www.cl.cam.ac.uk/~sjm217/papers/tor11datagramcomparison.pdf.

[18] NOWLAN, M. F., TIWARI, N., IYENGAR, J., AMIN, S. O., AND FORD, B. Fitting square pegs through round pipes. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012).

[19] NOWLAN, M. F., WOLINSKY, D., AND FORD, B. Reducing latency in Tor circuits with unordered delivery. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)* (2013).

[20] REARDON, J., AND GOLDBERG, I. Improving tor using a tcp-over-dtls tunnel. In *Proceedings of the 18th conference on USENIX security symposium* (2009), USENIX Association, pp. 119–134.

[21] TANG, C., AND GOLDBERG, I. An Improved Algorithm for Tor Circuit Scheduling. In *Proc. of the 17th Conference on Computer and Communication Security* (2010).

[22] WANG, T., BAUER, K., FORERO, C., AND GOLDBERG, I. Congestion-aware Path Selection for Tor. In *Proceedings of Financial Cryptography and Data Security (FC'12)* (February 2012).