# TCP Fast Open

Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain
Google
sivasankar@cs.ucsd.edu, {ycheng, hkchu, arvind}@google.com

Barath Raghavan
ICSI
barath@icsi.berkeley.edu

## ABSTRACT

Today's web services are dominated by TCP flows so short that they terminate a few round trips after handshaking; this handshake is a significant source of latency for such flows. In this paper we describe the design, implementation, and deployment of the TCP Fast Open protocol, a new mechanism that enables data exchange during TCP's initial handshake. In doing so, TCP Fast Open decreases application network latency by one full round-trip time, decreasing the delay experienced by such short TCP transfers.

We address the security issues inherent in allowing data exchange during the three-way handshake, which we mitigate using a security token that verifies IP address ownership. We detail other fall-back defense mechanisms and address issues we faced with middleboxes, backwards compatibility for existing network stacks, and incremental deployment. Based on traffic analysis and network emulation, we show that TCP Fast Open would decrease HTTP transaction network latency by 15% and whole-page load time over 10% on average, and in some cases up to 40%.

## Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network Communications; C.2.2 [**Network Protocols**]: TCP/IP

## General Terms

Design, Performance, Reliability, Security

## 1. INTRODUCTION

While web pages have grown significantly in recent years, network protocols have not scaled with them. Today's pages are on average over 300KB each, but most web objects are relatively small, with mean and median sizes of 7.3KB and 2.4KB respectively [25]. As a result of the preponderance of small objects in large pages, web transfer latency has come to be dominated by both the round-trip time (RTT) between the client and server and the number of round trips required to transfer application data. The RTT of a web flow largely comprises two components: transmission delay and propagation delay. Though network bandwidth has grown substantially over the past two decades thereby significantly reducing transmission delays, propagation delay is largely constrained by the speed of light and therefore has remained unchanged. Thus reducing the number of round trips required for the transfer of a web object is the most effective way to improve the latency of web applications [14, 18, 28, 31].

Today's TCP standard permits data exchange only after the client and server perform a handshake to establish a connection. This introduces one RTT of delay for each connection. For short transfers such as those common today on the web, this additional RTT is a significant portion of the flows' network latency [29]. One solution to this problem is to reuse connections for later requests (e.g. HTTP persistent connections [24]). This approach, while widely used, has limited utility. For example, the Chrome browser keeps idle HTTP 1.1 TCP connections open for several minutes to take advantage of persistent connections; despite this over one third of the HTTP requests it makes use new TCP connections. A recent study on a large CDN showed that on average only 2.4 HTTP requests were made per TCP connection [10]. This is due to several reasons as we describe in Section 2.

We find that the performance penalty incurred by a web flow due to its TCP handshake is between 10% and 30% of the latency to serve the HTTP request, as we show in detail in Section 2. To reduce or eliminate this cost, a simple solution is to exchange data during TCP's initial handshake (e.g. an HTTP GET request / response in SYN packets). However, a straightforward implementation of this idea is vulnerable to denial-of-service (DoS) attacks and may face difficulties with duplicate or stale SYNs. To avoid these issues, several TCP mechanisms have been proposed to allow data to be included in the initial handshake; however, these mechanisms were designed with different goals in mind, and none enjoy wide deployment due to a variety of compatibility and/or security issues [11, 12, 16].


ACM CoNEXT 2011, December 6–9 2011, Tokyo, Japan.

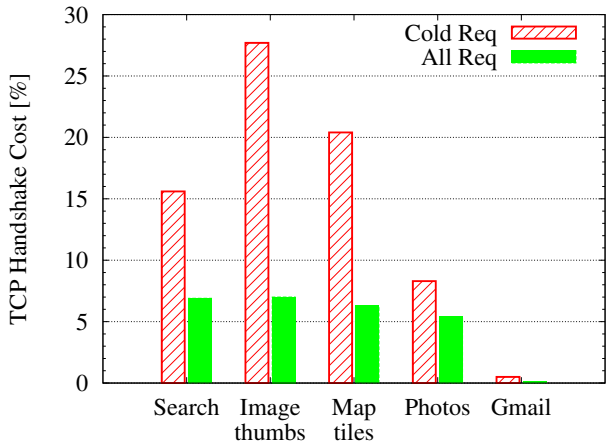


**Figure 1: TCP handshake time as a percentage of total HTTP request latency for Google.com. For the "All Req" category, handshake time is amortized over all HTTP requests for the Google service in question.**

In this paper we propose a new TCP mechanism called TCP Fast Open (TFO) that enables data to be exchanged safely during TCP's initial handshake. At the core of TFO is a security cookie that is used by the server to authenticate a client that is initiating a TFO connection. We describe the details of TFO, including how it exchanges data during the handshake, the protocol used for TFO cookies, and socket API extensions to enable TFO. In addition, we analyze the security of TFO and examine both the potential for new security vulnerabilities and their mitigation. We also describe our implementation of TFO in the Linux kernel and in the Chrome web browser and present the performance gains we see in our testbed experiments. Finally we examine deployment issues and related approaches.

## 2. MOTIVATION

Latency and page load time are important factors that influence user satisfaction with a website. Even small improvements in latency lead to noticeable increases in site visits and user satisfaction, and result in higher revenues [3, 6, 5]. While it is well known that small objects dominate web flows today, we sought to better understand the actual performance characteristics of today's flows and the performance bottlenecks they experience. To do so, we analyzed both Google web server logs and Chrome browser statistics to demonstrate that TCP's handshake is a key performance bottleneck for modern web transfers. Our intent is to highlight this practical problem through the analysis of large scale data and to estimate the potential benefits of TFO.

### 2.1 Google Server Logs Analysis

We begin by analyzing latency data from Google web server logs to study the impact of TCP's handshake on user-perceived HTTP request latency. We sampled a few billion HTTP requests (on port 80) to Google servers world-wide over 7 consecutive days in June 2011. These included requests to multiple Google services such as search, email, and photos. For each sampled request, we measured the latency from when the first byte of the request is received by the server to when the entire response is acknowledged. If the request is the first one of the TCP connection, this latency also includes the TCP handshake time since the browser needs to wait for the handshake to complete before sending the request. Note that our latency calculation includes both server processing time and network transfer time.

We define requests sent on new TCP connections as *cold requests* and those that reuse TCP connections as *warm requests*. We segregate requests by service and compute the fraction of time spent on TCP handshakes for cold requests. Similarly, we compute the amortized cost of TCP handshakes over both cold and warm requests for each service. The results shown in Figure 1 indicate that TCP handshakes account for 8% to 28% of the latency of cold requests for most services. Even the amortized cost for handshakes accounts for 5-7% of latency across both cold and warm requests, including photo services where the average response size is hundreds of kilobytes. (The only exception is Gmail because it downloads javascript upon a cold request and reuses the same connection for many subsequent warm requests.)

The cost of TCP handshakes is surprisingly high given that 92% of the requests that we see use HTTP/1.1 which supports persistent HTTP connections. Moreover, Google web servers keep idle connections open for several minutes. In theory, most requests should reuse existing TCP connections to minimize the penalty of a TCP handshake, but our analysis indicates this may not be happening. In order to understand if this problem persists for other web sites and what its cause(s) might be, next we analyze statistics from the Chrome web browser.

### 2.2 Chrome Browser Statistics

We processed Chrome browser statistics for 28 consecutive days in 2011; these only cover Chrome users who have opted into statistics collection and only contain anonymized data such as latency statistics. The statistics do however cover requests to all websites and not just Google services. Across billions of sampled HTTP latency records, we found that over 33% of requests made by Chrome are sent on newly created TCP connections even though it uses HTTP 1.1 persistent connections. The restricted effectiveness of persistent connections is due to several factors. Browsers today often open tens of parallel connections to accelerate page downloads, which limits connection reuse. Domain sharding or the placement of resources on different domains by content providers to increase parallelism in loading web pages also exacerbates this issue. In general, hosts and middle-boxes (NATs) also terminate idle TCP connections to minimize resource usage. The middle-box issue may be partly mitigated by using TCP keepalive probes, but this could be pro-

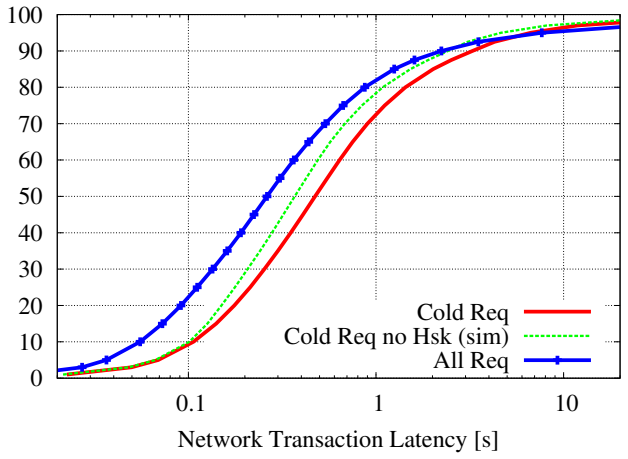


**Figure 2: CDF of the HTTP transaction network latency for Chrome Windows users. The Y-axis is the cumulative distribution of HTTP requests in percentiles. "Cold Req" and "Cold Req no Hsk (sim)" refer to requests that need to open new TCP connections, but the latter excludes TCP connect time. "All Req" refers to all requests, including both HTTP and HTTPS.**

hibitively power hungry on mobile devices [32]. Major mobile browsers close idle connections after mere seconds to conserve power.

To understand the latency impact of waiting for TCP's handshake to complete before transferring data, we plot the distribution of HTTP transaction network latency for cold requests and all requests in Figure 2. We measured network transaction latency from the time the browser schedules a request to the time it receives the entire response. If the browser does not have an idle TCP connection available to serve the request, it attempts to open a TCP connection. Thus TCP's handshake time is included in the network latency. Chrome also has a limit of 6 parallel connections per domain.

Figure 2 reveals that cold requests are often over 50% slower when compared to all requests in the same percentile. For example, the median latencies of cold requests and all requests are 549ms and 308ms, respectively. Many factors including DNS lookup, TCP slow-start, SSL handshake, and TCP handshake, may contribute to this slowdown. To isolate the cost of the TCP handshake, we plot network transaction latencies of cold requests excluding TCP handshake time.[1] This simulated distribution, labeled as "Cold Req no Hsk" in the figure, suggests that TCP handshake accounts for up to 25% of the latency between the 10th and 90th percentiles.

Thus the results of our analysis of both Google server logs and Chrome browser statistics suggest that sending an HTTP request and response during a TCP handshake can significantly improve HTTP transaction performance.[2]

[1]We measure TCP handshake time by the time it takes to finish the `connect()` system call in Chrome.

[2]We note that our estimates from Google server logs (which concern only requests for google.com) and Chrome browser statistics (which are across the web) differ likely because Google has a lower RTT and processing time than many other websites.

# 3. DESIGN

Our measurement results support the notion that eliminating one round trip from a web flow can provide immediate, measurable performance gains. However, it may be instructive to first consider the constraints we designed within and the assumptions we made while working on TCP Fast Open.

## 3.1 Context and Assumptions

The current TCP specification actually allows a client to include data in its SYN packet when initiating connections to servers, but forbids the servers from delivering the data to applications until the 3-way handshake (3WHS) completes [7]. Suppose for the moment that we were to remove this restriction, and simply enable ordinary TCP-based client applications to send HTTP GET requests in TCP SYN packets and servers to respond with data in their TCP SYN-ACK packets. While this would trivially meet the needs of TCP Fast Open, it would open the protocol up to a straightforward denial-of-service attack of both the server and arbitrary hosts: an attacker or set of attackers could send HTTP GET requests to a server while spoofing the source address of a victim host, thereby causing the server both to perform potentially expensive request processing and to send a potentially large response to a victim host. Thus we must build security mechanisms into TFO to protect both the server and other hosts from such attacks.

Our goal in designing TCP Fast Open was to enable each end of a TCP connection to safely transmit and process any received data while the 3WHS is still in progress. However, there are several other constraints that we kept in mind and assumptions that we were forced to make. For example, the TCP initial handshake is designed to deal with delayed or duplicate SYN packets received by a server and to prevent such packets from creating unnecessary new connections on the server; server applications are notified of new connections only when the first ACK is received from the client. We found that to manage stale or duplicate SYN packets would add significant complexity to our design, and thus we decided to accept old SYN packets with data in some rare cases; this decision restricts the use of TFO to applications that are tolerant to duplicate connection / data requests. Since a wide variety of applications can tolerate duplicate SYN packets with data (e.g. those that are idempotent or perform query-style transactions), we believe this constitutes an appropriate tradeoff.

Similarly, we make several assumptions about the setting in which TFO is deployed. We assume that servers cannot maintain permanent or semi-permanent per-client state since this may require too much server memory, and that servers may be behind load balancers or other such network devices. A stateless-server design is more desirable in this setting as it keeps state-management complexity to a minimum.

We also assume that servers cannot perform any operations to support TFO that are not reasonable to implement on the kernel's critical path (e.g. symmetric cryptography is possible, but asymmetric is not). We assume that clients are willing to install new software to support TFO and that small changes to applications are acceptable. Finally, we assume that it is acceptable to leverage other security mechanisms within a server's domain (if needed) in concert with TFO to provide the required security guarantees.

## 3.2 Design Overview

Our primary goal in the design of TFO is to prevent the source-address spoofing attack mentioned above. To prevent this attack, we use a security "cookie". A client that wishes to use TFO requests a cookie—an opaque bytestring—from the server in a regular TCP connection with the TFO TCP option included, and uses that cookie to perform fast open in subsequent connections to the same server. Figure 3 shows the usage of TFO. We begin by listing the steps a client performs to request a TFO cookie:

1. The client sends a SYN packet to the server with a Fast Open Cookie Request TCP option.
2. The server generates a cookie by encrypting the client's IP address under a secret key. The server responds to the client with a SYN-ACK that includes the generated Fast Open Cookie in a TCP option field.
3. The client caches the cookie for future TFO connections to the same server IP.

To use the fast open cookie that it received from a server, the client performs the following steps:

1. The client sends a SYN with the cached Fast Open cookie (as a TCP option) along with application data.
2. The server validates the cookie by decrypting it and comparing the IP address or by re-encrypting the IP address and comparing against the received cookie.
   (a) If the cookie is valid, the server sends a SYN-ACK that acknowledges the SYN and the data. The data is delivered to the server application.
   (b) Otherwise, the server drops the data, and sends a SYN-ACK that only acknowledges the SYN sequence number. The connection proceeds through a regular 3WHS.
3. If the data in the SYN packet was accepted, the server may transmit additional response data segments to the client before receiving the first ACK from the client.
4. The client sends an ACK acknowledging the server SYN. If the client's data was not acknowledged, it is retransmitted with the ACK.
5. The connection then proceeds like a normal TCP connection.

## 3.3 Cookie Design

The TFO cookie is an encrypted data string that is used to validate the IP ownership of the client. The server is responsible for generation and validation of TFO cookies. The client or the active-open end of a connection simply caches TFO cookies and returns these cookies to the server on subsequent connection initiations. The server encrypts the source IP address of the SYN packet sent by the client and generates a cookie of length up to 16 bytes. The encryption and decryption / validation operations are fast, comparable to the regular processing time of SYN or SYN-ACK packets.

Without the secret key used by the server upon cookie generation to encrypt the client's IP address, the client cannot generate a valid cookie. If the client were able to generate a valid cookie this would constitute a break of the underlying block cipher used for encryption. The server periodically revokes cookies it granted earlier by rotating the secret key used to generate them. This key rotation prevents malicious parties from harvesting many cookies over time for use in a coordinated attack on the server. Also, since client IP addresses may change periodically (e.g. if the client uses DHCP), revoking cookies granted earlier prevents a client from mounting an attack in which it changes its IP address but continues to spoof its old IP address in order to flood the new host that has that old address.

## 3.4 Security Considerations

TFO's goal is to allow data exchange during TCP's initial handshake while avoiding any new security vulnerabilities. Next we describe the main security issues that arise with TFO and how we mitigate them.

### *3.4.1 SYN Flood / Server Resource Exhaustion*

If the server were to always allow data in the SYN packet without any form of authentication or other defense mechanisms, an attacker could flood the server with spurious requests and force the server to spend CPU cycles processing these packets. Such an attack is typically aimed at forcing service failure due to server overload.

As noted earlier, TFO cookie validation is a simple operation that adds very little overhead on modern processors. If the cookies presented by the attacker are invalid, the data in the SYN packets is not accepted. Such connections fall back on regular TCP 3WHS and thus the server can be defended by existing techniques such as SYN cookies [19].

If the cookies that the attacker presents are valid—and note that any client can get a cookie from the server—then the server is vulnerable to resource exhaustion since the connections are accepted, and could consume significant CPU and memory resources on the server once the application is notified by the network stack. Thus it is crucial to restrict such damage.

To this end, we leverage a second mechanism: the server maintains a counter of total pending TFO connection requests either on a per service port basis or for the server

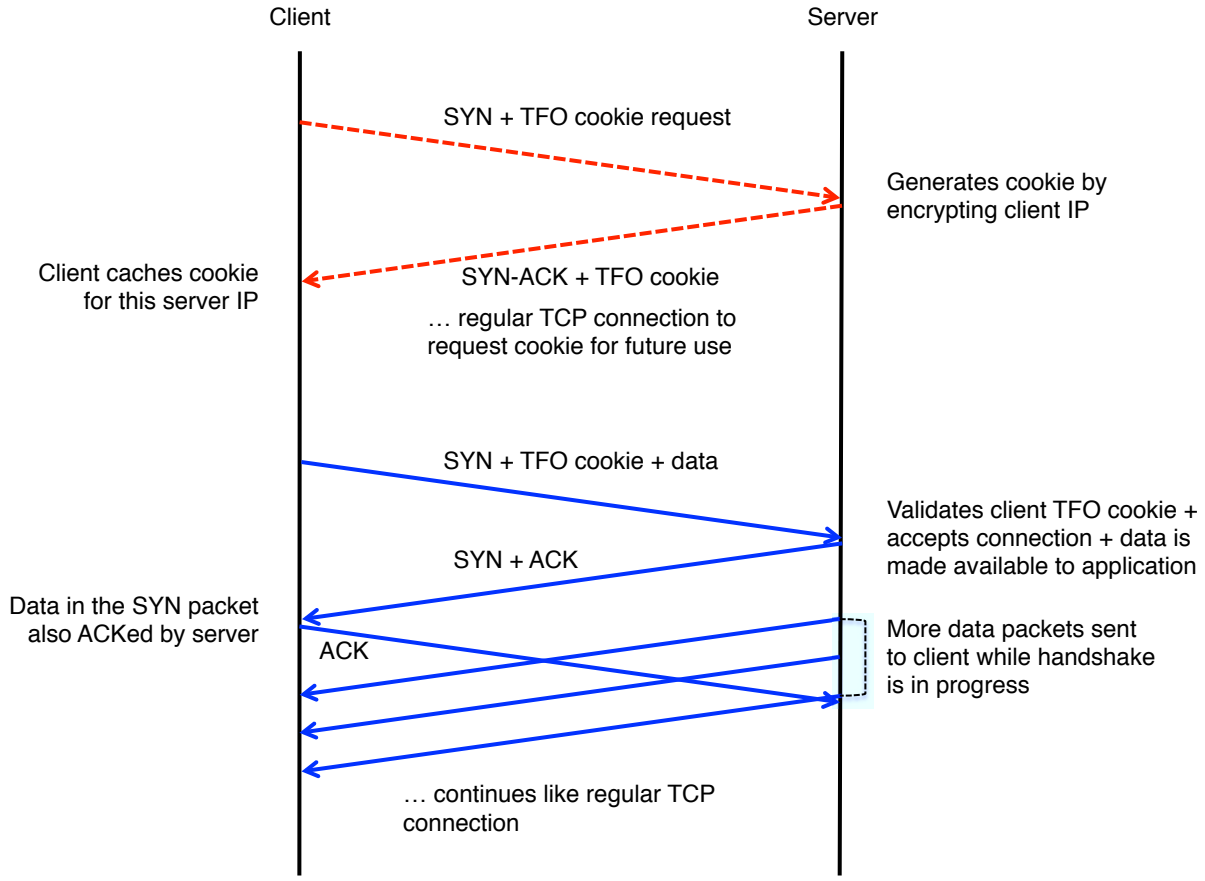


**Figure 3: TFO connection overview**

as a whole. This counter represents TFO connections that have been accepted by the server but that have not been migrated to the fully-established TCP state, which occurs only after receiving the first ACK from the peer (completion of 3WHS). When the number of pending TFO connections exceeds a certain threshold (that is administratively set), the server temporarily disables TFO and any incoming TFO requests fall back on regular 3WHS. This allows the usual SYN flood defense techniques [19] to prevent further damage until the pending TFO requests falls below the threshold. This limit makes it possible for an attacker to overflow the limit and temporarily disable TFO on the server, but we believe that this is unlikely to be of interest to an attacker since this would only disable the TFO "fast path" while leaving the service intact.

There is another subtle but important difference between TFO and a regular TCP handshake. When SYN flood attacks originally broke out in the late 1990s, they were aimed at overflowing the short SYN backlog queues on servers that were used to store information about incoming connection requests until the completion of 3WHS. In such an attack, the attacker sends a stream of SYN packets with spoofed source IP addresses until this SYN queue fills up. This causes new SYN packets to be dropped, resulting in service disruption. The attacker typically uses spoofed source IP addresses that are non-responsive; otherwise, the SYN-ACK would trigger a TCP RST from the host whose IP has been spoofed. The TCP RST packet would terminate the connection request on the server and free up the SYN queue, thereby defeating the attack.

With TFO, such RST packets from spoofed hosts would fuel the damage since the RST will terminate the pending TFO request on the server, allowing another spoofed TFO connection request to be accepted. To defend against this, any pending TFO connections that get terminated by a RST should continue to be counted against the threshold for pending TFO connections described previously until a timeout period has passed.

### *3.4.2 Amplified Reflection Attack*

While regular TCP restricts the response from a server to just one SYN-ACK packet, TFO allows the server to send a stream of data packets following the SYN-ACK to the source IP address of the SYN packet. If not for the TFO cookie, this could be used by an attacker to mount an amplified reflection attack against any victim of choice.

As mentioned in the previous section, the number of pending TFO connections on the server has a system limit, so the server is protected from resource exhaustion beyond the limit. This system limit also bounds the damage that an attacker can cause through reflection from that server. However, the attacker can still create a reflection attack from a large number of servers to a single victim host as follows.

First, the attacker has to steal or otherwise obtain a valid cookie from the target victim. This likely requires the attacker to compromise or collude with the victim host or network. If the victim host is already compromised, the attacker would likely have little value in mounting a reflection attack against the host itself, but the attacker might still be interested in mounting the attack to disrupt the compromised host's network. The stolen cookie is valid for a single server, so the attacker has to steal valid cookies from a large number of servers and use them before they expire to cause a noticeable impact on the compromised host's network.

We argue that if the attacker has already compromised the target network or host, then the attacker could directly start flooding the network from the compromised host without the use of a reflection attack. If servers still want to mitigate such an attack, one possible defense is to wait for the 3WHS to complete before sending data to the client. The server would still accept data in the SYN packet and allow the application to process it, but would make sure it is a valid connection—that is, 3WHS completes before sending the response to the client. For many applications this modification would yield little slowdown versus standard TFO as server processing time is often greater than the RTT of the connection.

## 3.5 Handling Duplicate SYN Segments

The current TCP standard allows SYNs to carry data but forbids delivering the data to the application until the connection handshake is completed in order to handle duplicate SYNs. Although TFO does not retransmit SYNs with data, it's possible that the network duplicates SYN packets with data, causing the data to be replayed at the server. Suppose a TFO client sends a SYN with data and actively closes the connection before the duplicate SYN arrives at the server. The server, being passively closed, does not retain any state about the closed connection, so accepts the duplicate SYN and processes (replays) the data. If the duplicate is generated within a 2MSL timeout, the server is likely to terminate the connection after receiving an RST since the client would process the server's SYN/ACK in the `TIME_WAIT` state. Nevertheless, the request will have been replayed.

One heuristic to address this is to extend the `LAST_ACK` state for 2MSL duration at the server (the passive close side) after receiving the final ACK from the client. This prevents some delayed duplicate packets from reaching the server application. Applications that are particularly intolerant to duplicate transactions, such as credit card transactions or banking applications, already have application-level measures to ensure idempotence. Alternatively, they can use a nonce to ensure that a transaction occurs only once. This challenge already exists today in another form: users often click refresh in web browsers if a page does not load quickly, resulting in duplicate transactions.

```
sd = socket(...);
bind(sd, ...);

int tfo_opt = 1;
setsockopt(sd, SOL_TCP, TCP_LISTEN_TFO,
       (void*)&tfo_opt, 4);

listen(sd, ...);
```

**Figure 4: Server application sample code.**

## 3.6 API Changes

One of our design goals was to avoid changes to socket libraries and to reuse existing APIs as much as possible. This minimizes changes to applications that wish to use TFO and poses less of a deployment hurdle.

The server-side API to use TFO is extremely simple. A server application enables TFO for incoming connections to a listening socket simply by enabling a new socket option. Figure 4 shows sample code to enable TFO on the server. The remaining socket calls (i.e. `listen()`, `accept()`, `send()`, `recv()`, etc.) remain unchanged.

On the client side, using TFO requires the application to provide a destination IP address and port number, as well as the data to send. The `sendto()` and `sendmsg()` system calls already provide such an interface; we extend them for use with TFO. When these system calls are used on a regular TCP socket, the destination address is ignored and they behave just like a `send()` call. When the new TFO flag is set, these calls are modified to initiate a TFO connection.

If the TFO cookie for the destination IP address is available, a SYN packet with the cookie and data is sent to initiate the TFO connection; the network stack handles this decision without the application's intervention. If the cookie is not available, it falls back on a regular TCP three-way handshake and the data is queued up for transmission when the 3WHS is completed. The SYN packet in this case also includes a TCP option requesting a TFO cookie from the server for later use. In general, the use and handling of TFO cookies is done by the networking stack and is completely transparent to the application. Therefore no API is needed to expose them. After the first `sendmsg()` or `sendto()` call, the rest of the socket calls from the application are unmodified.

The modified `sendto()` and `sendmsg()` calls return the number of bytes of data queued up in the kernel or sent in the SYN packet. They can be used with blocking or non-blocking sockets and their return values upon error are a combination of the error messages returned by `send()` and

```
sd = socket(...);

char msg[16] = "Hello TFO World";

send_len = sendto(sd, msg, 15, MSG_TFO,
                  server_addr, addr_len);
```

**Figure 5: Client application sample code. The `sendto()` call with `MSG_TFO` flag combines `connect()` and `send()` functionalities.**

`connect()`. Figure 5 shows sample code that a client application can use to connect to a server using TFO.

Besides these, one could imagine additional less critical APIs that could be provided to expose TFO information related to the connection, such as whether a connection was opened through a regular handshake or TFO, and whether a TFO attempt to a server succeeded; APIs to set TFO secret keys and flush the TFO cache might also prove useful.

The TFO cookie handling is transparent to applications and the cookies received by a client from a server are not directly readable by applications unless they have root privileges to sniff packets on the client. This prevents malicious sites from using simple browser hacks to trick users by making connections to other websites and stealing those TFO cookies for mounting an amplified reflection attack.

# 4. DEPLOYABILITY

Given that the main goal of TFO is improving the performance of short transfers such as the retrieval of web objects, being compatible with today's network architecture is key. Thus we designed TFO for incremental deployment. In doing so, we enabled it to gracefully fall back on standard TCP to ensure that current and future TCP connections can proceed in response to unexpected network events. In this section, we discuss the challenges of incremental deployability and our responses to these challenges.

## 4.1 New TCP options / data in SYN

Our primary deployment concern with TFO is regarding how Internet routers, middle-boxes, end-hosts, and other entities will handle new types of TCP packets such as those with new TCP options, SYN packets with data, and the like, as such packets are unusual in today's networks. One study found that some middle-boxes and hosts drop packets with unknown TCP options [23]. A more recent study found that 0.015% of the top 10,000 websites do not respond with a SYN-ACK after receiving a SYN with a non-standard or new TCP option [15]. Another study found that 6% of probed paths on the Internet drop SYN packets containing data [22].

If a SYN packet with TCP Fast Open option set does not elicit a response within the timeout period (regardless of where it is dropped), we simply retransmit the SYN without any data or non-standard TCP options. In doing so, TFO falls back on a regular TCP 3WHS and connectivity with the server is not lost. The client also caches the RTT to the server in its cookie cache and sets the SYN retransmit timeout to $1.5 \times RTT$, thereby reducing the ordinarily longer SYN timeout. If TFO fails repeatedly to a given server, the client remembers the server's IP address and disables TFO for that server in the future.

## 4.2 Server Farms

Given that many large web services place servers in server farms, another point of concern for us is how TFO would be used at such data centers. A common setup for server farms is for many servers to be behind a load balancer, sharing the same server IP. Client TCP connections are load balanced to different physical hosts, often without any stored state about previous connections from the same client IP. Clients cache the TFO cookie based on a server's IP; TFO connections from a particular client might be load balanced to a physical server different from the one that granted the TFO cookie. We use TFO in this setting by sharing a common secret key (used for encrypting and decrypting TFO cookies) among all the servers in the server farm. Secret key updates to the servers are made at about the same time on all the servers.

## 4.3 Network Address Translation (NAT)

Network Address Translation (NAT) is another challenge for TFO. Hosts behind a single NAT sharing the same public IP address are granted the same cookie for the same server; nevertheless, the clients can all still use TFO. However some carrier-grade NAT configurations use different public IP addresses for new TCP connections from the same client. In such cases, the TFO cookies cached by the client would not be valid and the server would fall back on a regular 3WHS and reject any data in the SYN packet. Despite this, since the server would reply with an ordinary SYN-ACK, the use of TFO in this scenario would not cause any latency penalty versus an ordinary TCP connection.

## 4.4 TCP Option Space

The availability of TCP option space in the SYN and SYN-ACK packets is an issue since many options are negotiated in these packets. We analysed the connections seen at Google's web servers and found that over 99% of incoming client connections had sufficient option space to accommodate a TFO cookie option with an 8 or 16 byte cookie in the SYN packets. Therefore, this is unlikely to be a concern for web traffic.

If other types of traffic use certain long TCP options (e.g. the TCP MD5 option), and space is insufficient in the TCP options field to accommodate the TFO cookie, the connection can safely fall back on regular 3WHS.

# 5. IMPLEMENTATION

We implemented TCP Fast Open in the Linux 2.6.34 kernel and are in the early stages of deploying it across Google. The entire kernel patch is about 2000 lines of code with

about 400 lines used for the client side TFO cookie cache. We also coordinated with the developers of Chrome to implement TFO support within the browser.

### 5.1 Kernel support

While the Linux TCP stack required fairly deep modification, a key aspect to both the design and implementation of TFO is that it *does not affect* TCP congestion control. That is, since congestion control only takes place after TCP's handshake completes, and TFO is only in use during the handshake, the two are entirely separate. Thus we did not have to modify any code relating to congestion control in the Linux kernel. Also note that the maximum number of data segments that a server can send before getting acknowledgements from the client is dictated by the initial congestion window and receiver window, but that neither of these values are affected by TFO. Our modifications included alterations to incoming packet handling in the LISTEN, SYN_SENT, and SYN_RCVD states and to the routines that transmit TCP packets (to include appropriate options as required for TFO).

Our implementation uses a fixed size, 8 byte TFO cookie. We use the 128-bit (16 byte) AES block cipher implementation available in the Linux Kernel CryptoAPI to encrypt each client IP value; we truncate the result to 8 bytes to generate the cookie. We pad IPv4 client IP addresses with zeros to create a 16 byte IP value while IPv6 addresses are used in full. To validate the cookie contained within an incoming TFO request, the server recomputes the 8 byte cookie value based upon the incoming source IP address and compares it to the cookie included by the client. The cookie generation and validation operations add cryptographic processing overhead on the server. Many modern processors include AES instructions in hardware and a single CPU core can support tens of thousands of 16-byte AES encryptions per second [20]. This is greater than the connection acceptance rate of many modern servers, and the processing overhead for this cryptographic operation is only a small fraction of typical connection processing time.

For the cookie cache—which is used by client hosts' network stacks—we implemented a simple LRU policy that caches cookies, RTT, and MSS by server IP. While we found that this policy worked well, this cache replacement policy is not in any way tied to the protocol.

### 5.2 Application support

Only small changes are required in user level applications. Server side applications need just a single additional line of code: a call to `setsockopt()` to set the TFO socket option for the listen socket. Client side applications must replace `connect()` and the first `send()` call with a single call to `sendto()` with the appropriate flags. In addition to using TCP Fast Open within our own custom socket programs, the Chrome web browser was also modified to use TFO, as was the web server with which we performed tests.

## 6. EVALUATION

In this section we evaluate the performance improvement conferred by TCP Fast Open in two contexts. First, we measure the whole-page download gains seen by a TFO-enabled Chrome browser visiting popular websites. Second, we measure the more surprising performance benefits of TFO on the server side.

### 6.1 Whole Page Download Performance

The primary goal of TFO is to eliminate one RTT of extra latency, thereby improving the performance of short flows. This is particularly important for cold HTTP requests. In Section 2, we estimated, based upon Chrome browser statistics, that TFO could improve HTTP transaction latency by up to 25%. Here we ask a more general question: how much does TFO speed up whole-page downloads? Unfortunately, this question does not have a straightforward answer. On average, major web pages consist of 44 resources distributed across 7 different domains [25], and modern browsers have complex scheduling routines to fetch these resources using multiple parallel TCP connections.

First, we benchmarked several popular websites from the Alexa top 500 websites list [1]. The testbed for these experiments consists of a single machine (Intel Core 2 Quad CPU 2.4GHz, 8GB RAM) that runs our TFO-enabled Linux kernel and Chrome browser. We used the Google web page replay tool to benchmark the web page download latency for TFO-enabled Chrome and for standard Chrome [4]. The page replay tool has two modes: record and replay. In record mode, the tool passively records all DNS and TCP traffic sent from and received by the browser into a local database. In replay mode, the tool runs a DNS server on the local machine and redirects the browser's HTTP requests to the local proxy run by the tool. The replay can also leverage dummynet to emulate different network delays, bandwidths, and random packet loss [26]. All connections during replay use the loopback interface with a reduced MTU of 1500 bytes.

In our experiment, we emulated a broadband user with 4Mbps downlink and 256Kbps uplink bandwidth and with a 128KB buffer; this is a popular configuration as found by Netalyzer [21]. First, we used the tool to record the home pages of Amazon.com, the New York Times, the Wall Street Journal, and the Wikipedia page for TCP[3]. We then replayed each web page 20 times with and without TFO support in Chrome, and did so with three different RTTs: 20ms, 100ms, and 200ms. Therefore, for each page we gathered 120 samples. For each replay we performed a cold start of the browser and used a new user configuration folder (with an empty cache) to avoid caching effects and persistent connections to the replay tool's proxy. Since all connections use the loopback interface, the TFO-enabled browser always has a valid TFO cookie and thus sends the cold HTTP requests in the SYN packet.

[3]http://en.wikipedia.org/wiki/Transmission_Control_Protocol

| Page | RTT(ms) | PLT : non-TFO (s) | PLT : TFO (s) | Improv. |
|---|---|---|---|---|
| amazon.com | 20 | 1.54 | 1.48 | 4% |
| | 100 | 2.60 | 2.34 | 10% |
| | 200 | 4.10 | 3.66 | 11% |
| nytimes.com | 20 | 3.70 | 3.56 | 4% |
| | 100 | 4.59 | 4.30 | 6% |
| | 200 | 6.73 | 5.55 | 18% |
| wsj.com | 20 | 5.74 | 5.48 | 5% |
| | 100 | 7.08 | 6.60 | 7% |
| | 200 | 9.46 | 8.47 | 11% |
| TCP wikipedia page | 20 | 2.10 | 1.95 | 7% |
| | 100 | 3.49 | 2.92 | 16% |
| | 200 | 5.15 | 3.03 | 41% |

**Table 1: Average page load time (PLT) in seconds for various pages for an emulated residential broadband user with a 4Mbps/256Kbps link. In all tests, the standard deviations of the PLT are within 5% of the average except for amazon.com with 20ms RTT (7%).**

Thus for each replay with TFO-enabled browsing, we emulated a user with an empty browser cache visiting the website with TFO-enabled servers. For each page, the browser reports the page load time (PLT) that is measured from when the browser starts processing the URL until the browser *onload event* begins [30].[4] The PLT includes HTTP redirects, accessing the local cache, DNS lookups, HTTP transactions, and processing the root document. The results of the replays are shown in Table 1. As expected, TFO improves the PLT when the RTT is high for all the sites we tested. When the RTT is small and the network delay is only a small fraction of PLT, the resource processing time would exceed network time, so the gains from TFO are expected to be small. But even for pages heavy on content and with short emulated RTT (i.e. 20ms), TFO accelerates PLT by 4–5%. Conversely, for simpler pages such as wikipedia, the browser spends most of its time waiting for network transfers rather than processing the retrieved content, and thus TFO offers significant improvements of 16% and 41% with 100ms and 200ms RTTs respectively. The 200ms RTT figures roughly correspond to the expected performance on mobile devices since mobile RTTs are typically on the order of 100–200ms [27].

## 6.2 Server performance

To measure the impact of TFO on the server, we wrote a client program that generates HTTP 1.0 requests at a constant rate to an Apache server and fetches a 5KB web page. We used client and server machines with configurations similar to that in Section 6.1 and connected them through a Gigabit Ethernet switch; the RTT between them was about 100$\mu$s. We limited the server machine to only use one CPU core and measured the CPU utilization using OProfile to evaluate the overhead added by cookie generation/validation operations and other TFO related processing. At each request rate, we measured the average CPU utilization across four 5 minute long trials for regular TCP and TFO. Figure 6 shows that server CPU utilization is nearly the same with and without TFO—in fact the CPU utilization was marginally lower when using TFO between 2000 and 5000 requests per second. We attribute this to the fewer packets that the server has to process when TFO is enabled as the request is included in the SYN packet. The AES encryption function used for cookie validation accounted for less than 0.3% of CPU utilization even at 6000 requests per second.

To further stress the connection setup process, we created another client load generator that repeatedly makes TFO connections to the server and requests the default Apache home page. This server response fits within a single response packet thereby making the connection handshake a significant fraction of the entire connection. The client program operates in

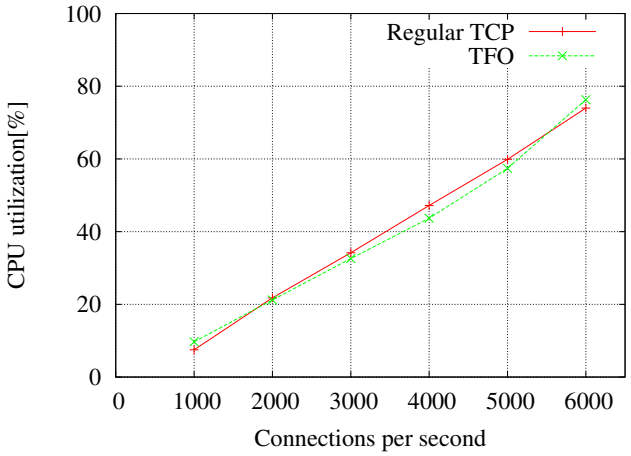


**Figure 6: CPU utilization vs. web server load level**

[4]To extract the PLT from Chrome, we opened the browser's javascript console and entered "performance.timing.loadEventEnd - performance.timing.navigationStart".

a closed loop and maintains one outstanding request to the server at any time; it creates a new connection to the server and sends another request as soon as it receives one complete HTTP response. With regular TCP, the server was able to sustain an average rate of 2876.4 transactions per second. Surprisingly, with the use of TFO, the server's sustained rate rose to 3548.7 transactions per second. This result is likely due to several factors: (1) one RTT saved per request, (2) fewer CPU cycles spent by the server to process each request (as the request is received in the SYN packet itself), and (3) one system call saved per request on the client.

# 7. DISCUSSION

Over the past year that we have spent discussing, designing, and implementing TCP Fast Open we have considered several alternative approaches to the design of TFO cookies, to the semantics of TFO, and to server-side attack mitigation. Here we consider those approaches and discuss their benefits and drawbacks.

## 7.1 One Time Cookies

To prevent attacks in which a host reuses a cookie or cookies that it collects either legitimately or illegitimately, next we discuss an alternative design approach that we considered but ultimately did not implement (largely to keep the mechanism as simple as possible). In this approach, each TCP Fast Open cookie is valid for only one Fast Open. A client that wants to do more than one Fast Open must request more cookies to perform those subsequent Fast Opens. All open TCP connections (regardless of how they were opened) would have a limitation that they can only issue one Fast Open cookie for the lifetime of the connection, and that a cookie cannot be issued until the server has received at least one ACK from the client—this maintains a one-to-one relationship between the number of currently valid cookies issued for a client-server pair and the number of TCP handshakes the pair have completed at some time.

Thus, a client host would a) open a connection with a normal three-way handshake, b) request a one-time Fast Open cookie, c) proceed as usual with the connection and eventually close it, d) open a new connection using its Fast Open cookie, and e) request a new cookie during this new connection. Clients that wish to open parallel TFO connections to a server would acquire multiple cookies to the same server across multiple regular TCP handshakes. In the (client) kernel, this approach would change the abstraction slightly, from a one-to-one mapping from server IP to cookie to a set mapping of server IP to a set of cookies; this change would not affect applications.

To implement one-time cookies, the zero-padding used for IPv4 addresses would be replaced by a 64-bit unsigned integer counter during cookie computation, thus the cookie would be the encryption of the concatenation of the server IP, client IP, and counter. This ensures that each cookie is unique, even for the same client-server pair. In this design, there need only be one counter per server, which is incremented whenever a Fast Open cookie is issued to any host.

There are several methods that could be used to prevent cookie reuse. Standalone servers would keep a lookup table to make sure that a cookie isn't reused in some small time window (e.g. a few minutes), and if it is, the server would fall back on a normal handshake. For servers behind load balancers, the load balancers could either do the same or, alternatively, could always hash the cookie value consistently to a destination backend server, thereby ensuring that if a cookie is reused then the same server will receive the duplicate requests, so the client(s) will be caught. If no load balancer modification is possible—as may be the case for large production web services—two options are possible: a) the service simply allows for a cookie to be reused $n$ times where $n$ is the number of servers behind a load balancer or b) the servers behind a load balancer periodically exchange information about recently seen cookies.

While this one-time cookie approach is more complex, it may have the benefit of thwarting some amplification and resource exhaustion attacks. Standalone servers or server farms with load balancers modified as described above would also have another benefit of providing TCP's usual semantics and not be exposed to the duplicate SYN issue since the cookie in a duplicate SYN would be rejected.

## 7.2 Data after SYN

Some applications may require the transmission of initial data requests that cannot fit in a single packet. Thus the room provided by TFO in the SYN packet may be insufficient. Our TFO protocol design can easily support transmission of additional data packets following a TFO-enabled SYN packet (before receiving a SYN-ACK from the server). However these data packets would have the ACK flag unset since the initial sequence number of the server is unknown until the receipt of the SYN-ACK. Our experiments revealed that Internet paths originating at several major ISPs drop data packets without the ACK flag. In order to not introduce any additional deployment constraints, we decided to disallow data packets sent by the client following a TFO-enabled SYN packet. This effectively limits the amount of data to be sent by the client during 3WHS to a single MSS; all this data must fit within the initial SYN packet. This is sufficient for many client applications such as HTTP web requests. The server is not limited in this way, and thus will be able to send up to what the advertised receive window in the client's SYN packet and TCP's initial congestion window allow.

## 7.3 Server side TFO cache

TFO includes a counter of total pending TFO connection requests on a per service port basis or for the whole server. Therefore it is possible for an attacker to force the server to disable TFO for all clients by flooding the server with spurious TFO requests using a cookie it obtained itself or using a stolen cookie from a compromised host.

The server can avoid disabling TFO for all clients by maintaining a small cache of recently received TFO connection requests from different client IP addresses. For each client IP address in the cache, the server stores the number of pending TFO connection requests from the client IP. If the pending TFO requests from a particular client IP exceeds the administratively set threshold, the server can selectively disable TFO for just that client IP address. The cache can store, for example, 10,000 client IP addresses using a modest amount of memory. The cache uses an LRU replacement policy.

The server would still have to maintain the global or listener port-level accounting to serve as the final defense, but smaller thresholds may be used for individual client IP addresses. This is because a large number of compromised hosts can mount a coordinated attack in which they overflow the server-side cache and thus the cache entries replaced are those with the client IP addresses of other compromised hosts which are also flooding the server with spurious requests. Each client IP does not exceed the IP-level pending request threshold before its entry gets evicted from the cache, but it soon sends more spurious requests and is added to the cache again with its pending-requests counter reset. The server side cache increases the number of valid cookies that the attacker must steal to disable TFO for everyone, but does not completely eliminate the possibility.

## 8. RELATED WORK

Several instances of prior work aim to improve TCP performance by directly eliminating the three-way handshake, or more generally by designing server-stateless extensions. Here we attempt to place TCP Fast Open in context and compare the design tradeoffs that motivated prior work and motivate our work.

TCP Extensions for Transactions (T/TCP), among its other features, bypasses TCP's connection handshake, and thus shares both the goals and the challenges of TFO [16]. T/TCP focuses its effort on combating old or duplicate SYNs, and does not aim to mitigate security vulnerabilities introduced by bypassing 3WHS. Its TAO option and connection count add complexity and require the server to keep state per remote host, while still leaving it open for attack. It is possible for an attacker to fake a congestion control value that will pass the TAO test. Ultimately its scheme is insecure, as discussed by prior analyses [9, 8].

As we noted earlier, our focus with TFO is on its security and practicality, and thus we made the design decision to allow old, duplicate SYN packets with data. We believe this approach strikes the right balance, and makes TFO much simpler and more appealing to TCP implementers and application writers. While TFO's vulnerability to SYN flood attacks is no different from unmodified TCP, the damage an attacker can inflict upon the server may be much worse, and thus deserves careful consideration. Numerous prior studies discuss approaches to mitigating ordinary SYN flood attacks (that is, floods of SYNs without data) [19]. However, none of these approaches, from stateless solutions such as SYN-cookies to stateful solutions such as SYN Caches, can preserve data sent along with SYNs while providing an effective defense. Thus we concluded that the best defense is simply to disable TFO when a host is suspected to be under a SYN flood attack (e.g. when the SYN backlog is filled). Once TFO is disabled, normal SYN flood defenses can be employed.

Like TFO, TCPCT also allows SYN and SYN-ACK packets to carry data, though TCPCT is primarily designed to eliminate server state during the initial handshake and to defend from spoofed DoS attacks [11]. Therefore, TCPCT and TFO are designed to meet different needs and are not directly comparable. A TCPCT-enabled server does not keep any connection state during TCP's initial handshake, and thus the server-side application must consume the data in the SYN and immediately produce the response data to be included in SYN-ACK. Otherwise, the application's response is forced to wait until the handshake completes. This approach also constrains the application's response size to only one packet. By contrast, TFO allows the server to respond to data during the handshake even after the SYN-ACK is sent.

A recent proposal, Rapid-Restart [12], was proposed after the TFO IETF draft and has similar goals. Rapid-Restart is based on TCPCT; both the server and the client cache TCP control blocks after a connection is terminated, deviating from TCPCT's original design goal of saving server memory. The client sends a SYN with data and the previously stored TCPCT cookie. The server accepts the connection if the cookie and the IP match its cached copies. Rapid-Restart does not scale because it requires per-connection state at the server. Moreover, Rapid-Restart cannot be used in server farms because connection state is retained only by the server that processed the last connection from the client, and a subsequent connection from that client may be directed to a different server in the farm unless the load balancer is modified.

More recently Zhou *et al.* proposed ASAP which provides a solution to reduce DNS and eliminate TCP handshake latency [33]. It employs public-key certificates issued by a provenance verifier and signed by clients to ensure authenticity of requests to a server. In doing so, it offers more generality at the expense of computational overhead and incremental deployability.

Since none of the proposals we have discussed above are deployed, browser vendors have developed their own feature - "PRECONNECT" to avoid TCP handshake latency. Chrome and Internet Explorer 9 maintain a history of the domains for frequently visited web pages. The browsers then speculatively pre-open TCP connections to these domains before the user initiates any requests for them. Tests show this feature improves overall page load time by 6-10% for the top 35 websites [2, 13]. The downside of this approach is that it wastes server and network resources by initiating and maintaining idle connections due to mis-speculation; the hit rate for these mechanisms is fairly low. TFO offers similar performance improvement without the added overhead.

## 9. SUMMARY

To improve the performance of short transfers, we proposed TCP Fast Open (TFO), which enables data to be exchanged safely during TCP's initial handshake. Our analysis of both Google server logs and Chrome browser statistics shows that handshaking has become a performance bottleneck for web transfers. TFO enables applications to decrease request latency by one round-trip time while avoiding severe security ramifications. At the core of TFO is a security cookie issued by the server to authenticate clients that initiate TFO connections. We believe that this cookie mechanism provides an acceptable defense against potential denial of service attacks. TFO is also designed to fall back gracefully on regular TCP handshaking as needed.

Our goal—of including data in TCP SYN and SYN-ACK packets—is not novel. The TCP standard already allows it, but forbids the receiver from processing the data until the handshake completes. Several recent proposals achieve similar goals to TFO but have not seen wide deployment. The main contribution of TFO is the simplicity of its design, allowing rapid and incremental deployment while maintaining reasonable defense against denial-of-service attacks. We believe TFO interoperates well with existing TCP implementations, middle-boxes, server farms, and legacy server and client applications.

We have implemented TFO in the Linux kernel and shown that it imposes minimal performance overhead for clients and servers, with significant latency improvement for short transfers. Our analysis and testbed results show that TFO can improve single HTTP request latency by over 10% and the overall page load time from 4% to 40%. We have submitted an Internet draft to the IETF [17] and are in the process of deploying TFO on Google servers. We are also publishing our implementation for inclusion in the Linux kernel.

## 10. ACKNOWLEDGMENTS

We thank the CoNEXT reviewers and our shepherd Kyoungsoo Park for their comments on the paper. We are especially thankful to Mike Belshe for motivating this work and making the Chrome browser an early adopter of TFO. We would also like to thank Adam Langley, Tom Herbert, Roberto Peon, and Mathew Mathis for their insightful comments on early designs of TFO.

## 11. REFERENCES

[1] Alexa top 500 global sites. http://www.alexa.com/topsites.
[2] Chromium by "pre-connect" to accelerate web browsing. http://goo.gl/KPoNW.
[3] The need for speed. http://www.technologyreview.com/files/54902/GoogleSpeed_charts.pdf.
[4] Web page replay tool. http://code.google.com/p/web-page-replay/.
[5] Web performance and ecommerce. http://www.strangeloopnetworks.com/resources/#Infographics.
[6] When seconds count. http://www.gomez.com/wp-content/downloads/GomezWebSpeedSurvey.pdf.
[7] Transmission Control Protocol, 1981. RFC 793.
[8] Security problems associated with T/TCP. http://www.mid-way.org/doc/ttcp-sec.txt, 1996.
[9] T/TCP Vulnerabilities. *Phrack Magazine*, 8(53), July 1998.
[10] M. Al-Fares, K. Elmeleegy, B. Reed, and I. Gashinsky. Overclocking the yahoo! cdn for faster web page loads. In *Proceedings of the 11th Conference on Internet Measurement (IMC) (To Appear)*, 2011.
[11] W. Allen Simpson. TCP Cookie Transactions (TCPCT), 2011. RFC 6013.
[12] W. Allen Simpson. Tcp cookie transactions (tcpct) rapid restart, June 2011. Work in progress.
[13] M. Belshe. The era of browser preconnect. http://www.belshe.com/2011/02/10/the-era-of-browser-preconnect/.
[14] M. Belshe. More bandwidth doesnt matter (much). http://www.belshe.com/2010/05/24/more-bandwidth-doesnt-matter-much/.
[15] A. Bittau, M. Hamburg, M. Handley, D. Maziers, and D. Boeh. The case for ubiquitous transport-level encryption. In *Proceedings of the 19th USENIX Security Symposium*, August 2010.
[16] R. Braden. T/TCP – TCP Extensions for Transactions Functional Specification, 1994. RFC 1644.
[17] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. Tcp fast open. http://www.ietf.org/id/draft-cheng-tcpm-fastopen-00.txt, March 2011. Work in progress.
[18] N. Dukkipati et al. An argument for increasing tcp's initial congestion window. *SIGCOMM Computer Communication Review*, 40(3), July 2010.
[19] W. Eddy. TCP SYN Flooding Attacks and Common Mitigations, 2007. RFC 4987.
[20] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.
[21] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: illuminating the edge network. In *Proceedings of the 10th Conference on Internet measurement (IMC)*, 2010.
[22] A. Langley. Probing the viability of tcp extensions. http://www.imperialviolet.org/binary/ecntest.pdf.
[23] A. Medina, M. Allman, and S. Floyd. Measuring interactions between transport protocols and middleboxes. In *Proceedings of the 4th Conference on Internet Measurement (IMC)*, 2004.
[24] J. C. Mogul. The case for persistent-connection http. *SIGCOMM Comput. Commun. Rev.*, 25:299–313, October 1995.
[25] S. Ramachandran. Web metrics: Size and number of resources. http://code.google.com/speed/articles/web-metrics.html.
[26] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27:31–41, Jan 1997.
[27] P. Romirer-Maierhofer, F. Ricciato, A. DAlconzo, R. Franzan, and W. Karner. Network-wide measurements of TCP RTT in 3g. In *Traffic Monitoring and Analysis*, volume 5537 of *Lecture Notes in Computer Science*, pages 17–25. Springer Berlin / Heidelberg, 2009.
[28] P. Sun, M. Yu, M. J. Freedman, and J. Rexford. Identifying performance bottlenecks in CDNs through TCP-level monitoring. In *Proceedings of Workshop on Measurements Up the Stack (To appear)*, August 2011.
[29] J. Touch, J. Heidemann, and K. Obraczka. Analysis of HTTP performance. December 1998.
[30] Z. Wang. Navigation timing. http://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html.
[31] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones. In *Proceedings of Hot Mobile Workshop*, March 2011.
[32] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proceedings of ACM SIGCOMM 2011*.
[33] W. Zhou, Q. Li, M. Caesar, and B. Godfrey. ASAP: A low-latency transport layer. In *Proceedings of ACM CoNEXT 2011*.