The use of TLS in Censorship Circumvention

Sergey Frolov
University of Colorado Boulder
sergey.frolov@colorado.edu

Eric Wustrow
University of Colorado Boulder
ewust@colorado.edu

***Abstract*—TLS, the Transport Layer Security protocol, has quickly become the most popular protocol on the Internet, already used to load over 70% of web pages in Mozilla Firefox. Due to its ubiquity, TLS is also a popular protocol for censorship circumvention tools, including Tor and Signal, among others.**

However, the wide range of features supported in TLS makes it possible to distinguish implementations from one another by what set of cipher suites, elliptic curves, signature algorithms, and other extensions they support. Already, censors have used deep packet inspection (DPI) to identify and block popular circumvention tools based on the fingerprint of their TLS implementation.

In response, many circumvention tools have attempted to mimic popular TLS implementations such as browsers, but this technique has several challenges. First, it is burdensome to keep up with the rapidly-changing browser TLS implementations, and know what fingerprints would be good candidates to mimic. Second, TLS implementations can be difficult to mimic correctly, as they offer many features that may not be supported by the relatively lightweight libraries used in typical circumvention tools. Finally, dependency changes and updates to the underlying libraries can silently impact what an application's TLS fingerprint looks like, making it difficult for tool maintainers to keep up.

In this paper, we collect and analyze real-world TLS traffic from over 11.8 billion TLS connections over 9 months to identify a wide range of TLS client implementations actually used on the Internet. We use our data to analyze TLS implementations of several popular censorship circumvention tools, including Lantern, Psiphon, Signal, Outline, TapDance, and Tor (Snowflake and meek pluggable transports). We find that the many of these tools use TLS configurations that are easily distinguishable from the real-world traffic they attempt to mimic, even when these tools have put effort into parroting popular TLS implementations.

To address this problem, we have developed a library, uTLS, that enables tool maintainers to *automatically* mimic other popular TLS implementations. Using our real-world traffic dataset, we observe many popular TLS implementations we are able to correctly mimic with uTLS, and we describe ways our tool can more flexibly adapt to the dynamic TLS ecosystem with minimal manual effort.

I. Introduction

The Transport Layer Security (TLS) protocol is quickly becoming the most popular protocol on the Internet, securing network communication from interference and eavesdropping. Already, 70% of page loads by Firefox users make use of TLS [54], and adoption continues to grow as more websites, services, and applications switch to TLS.

Given the prevalence of TLS, it is commonly used by circumvention tools to evade Internet censorship. Because censors can easily identify and block custom protocols [30], circumvention tools have turned to using existing protocols. TLS offers a convenient choice for these tools, providing plenty of legitimate cover traffic from web browsers and other TLS user, protection of content from eavesdroppers, and several libraries to choose from that support it.

However, simply using TLS for a transport protocol is not enough to evade censors. Since TLS handshakes are not encrypted, censors can identify a client's purported support for encryption functions, key exchange algorithms, and extensions, all of which are sent in the clear in the first *Client Hello* message.

In fact, popular tools such as Tor have already been blocked numerous times due to its distinctive SSL/TLS features [49], [1], [42], [13], [60], [50], [51]. Even tools that successfully mimicked or tunneled through other popular TLS implementations have suffered censorship. For example, in 2016, Cyberoam firewalls were able to block meek, a popular pluggable transport used in Tor to evade censors, by fingerprinting its TLS connection handshake [21]. Although meek used a genuine version of Firefox bundled with Tor, this version had become outdated compared to the rest of the Firefox user population, comprising only a 0.38% share of desktop browsers, compared to the more recent Firefox 45 comprising 10.69% at the time [53]. This allowed Cyberoam to block meek with minimal collateral damage.

The problem was temporarily corrected by updating to Firefox 45, but only a few months later, meek was blocked again in the same manner, this time by the FortiGuard firewall, which identified a combination of SNI extension values sent by meek and otherwise matching the signature of Firefox 45 [22]. At that time, Firefox 47 had been released, supporting a distinguishable set of features. The rapid pace of new implementations and versions is a difficult task to keep up with.

Another motivating example of these challenges is found in the Signal secure messaging application [57]. Until recently, Signal employed domain fronting to evade censorship in several countries including Egypt, Saudi Arabia, and the United Arab Emirates [25], [41]. However, due to a complicated interaction with the library it used to implement TLS, we find that Client Hello messages sent by Signal while domain fronting differ from their intended specification, ultimately allowing them to be distinguished from the implementations

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23511
www.ndss-symposium.org

they attempted to mimic and easy for censors to block[1].

These examples demonstrate the difficulties in making TLS implementations robust against censorship. To further study this problem, we collect and analyze real-world TLS handshakes, and compare them to handshakes produced by several censorship circumvention tools. Our study examines TLS connections from a 10 Gbps tap at the University of Colorado Boulder, serving over 33,000 students and 7,000 faculty. We collected over 11 billion TLS connections over a 9 month period. For each connection, we generate a hash (fingerprint) [47], [34] over unchanging parts of the Client Hello message, allowing us to group connections that were made by the same implementation together. We also collect information on corresponding Server Hello messages, and anonymized SNI and destination IPs to assist further analysis.

Using our data, we find several problems across many circumvention tools we analyze, including Signal, Lantern, and Snowflake, and uncover less serious but still problematic issues with Psiphon and meek. To enable other researchers to use our dataset, we have released our data through a website, available at **https://tlsfingerprint.io**.

To address the challenge faced by existing circumvention tools, we have developed a client TLS library, *uTLS*, purpose built to provide fine-grained control over TLS handshakes. uTLS allows developers to specify arbitrary cipher suites and extensions in order to accurately mimic other popular TLS implementations. Moreover, we integrate our dataset with uTLS to allow developers to copy automatically-generated code from our website to configure uTLS to mimic popular fingerprints we have observed.

We describe and overcome several challenges in correctly mimicking implementations, and we implement multiple evasion strategies in uTLS including mimicry and randomized fingerprints, and finally evaluate each of these strategies using our dataset. In addition, we have worked with several existing circumvention tools to integrate our uTLS library into their systems.

In our data collection, we have made sure to collect only aggregates of potentially sensitive data to protect user privacy. We have applied for and received IRB exemption from our institution for this study, and worked closely with our institution's networking and security teams to deploy our system in a way that protects the privacy of user traffic. Our findings were disclosed responsibly to the projects and tools impacted by our results.

Our contributions are as follows:

- We collect and analyze over 11 billion TLS Client Hello messages over a 9 month period, as well as 5.9 billion TLS Server Hellos over several months. We intend to continue collecting future data.
- We analyze existing censorship circumvention projects that use or mimic TLS, finding that many are trivially identifiable in practice, and potentially at risk of being blocked by censors.
- We develop a library, *uTLS*, that allows developers to easily mimic arbitrary TLS handshakes of popular implementations, allowing censorship circumvention tools to better camouflage themselves against censors. We use our collected data to enhance uTLS, allowing *automated mimicry* of popular TLS implementations.
- We release our dataset through a website[2], allowing researchers to browse popular TLS implementation fingerprints, search for support of ciphers, extensions, or other cryptographic parameters, and compare the TLS fingerprints generated by their own applications and devices.

We present background in Section II, the design of our collection infrastructure in Section III, and high level results from our dataset as it pertains to censorship in Section IV. We present our analysis and findings on circumvention tools in Section V, and present defenses and lessons in Section VI. We go on to describe our uTLS library in Section VII, discuss future and related work in Sections IX and X, and finally conclude in Section XI.

II. Background

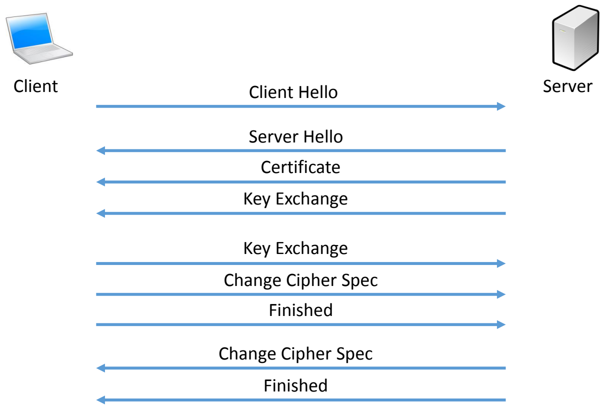

Fig. 1. **TLS Handshake** — The TLS handshake contains several messages sent unencrypted, including the Client Hello. This allows us to fingerprint client implementations by the features and parameters they send in this initial message.

TLS typically takes place over a TCP connection between a client and server[3].

After a TCP connection is established, the client and server perform a TLS handshake that allows them to authenticate identities and agree on keys, ciphers, and other cryptographic parameters to be used in the connection. The remainder of the connection is encrypted with the agreed upon methods and secrets. Figure 1 shows an overview of the TLS 1.2 handshake.

The first message in the TLS handshake is a Client Hello message. This message is sent in the clear, and allows the client to specify features and parameters that it supports. This includes what versions of TLS the client supports, a list of supported cipher suites and compression methods, a random

[1] Signal has since phased out domain fronting for unrelated reasons

[2] https://tlsfingerprint.io

[3] Although TLS can happen on top of other protocols (such as UDP), for the purposes of this paper, we focus on TLS over TCP.

nonce to protect against replay attacks, and an optional list of extensions. Extensions allow clients and servers to agree on additional features or parameters. While there are over 20 extensions specified by various TLS versions, we pay extra attention to the contents of few of them in this paper, which we use as part of the Client Hello fingerprint.

Server Name Indication (SNI) allows a client to specify the domain being requested in the Client Hello, allowing the server to send the correct certificate if multiple hosts are supported. As this is sent before the handshake, SNIs are sent unencrypted.

Supported Groups This extension specifies a list of supported mathematical groups that the client can use in key exchanges and for authentication. For example, the client can specify support for groups such as `x25519`, `secp256k1` to specify support for Curve25519 and the NIST P-256 Koblitz curve for use in ECDHE key exchanges.

Signature Algorithms Clients can specify combinations of hash and signature algorithms they support for authenticating their peers. Traditionally these have come in the form of a signature and hash algorithm pair, such as `rsa_sha256` or `ecdsa_sha512`. More recently, signature algorithms have been expanded to specify alternate padding schemes (such as RSA PSS).

Elliptic Curve Point Format specifies the encoding formats supported by the client when sending or receiving elliptic curve points.

Application Layer Protocol Negotiation (ALPN) allows clients to negotiate the application protocols they support on top of TLS. For instance, a web browser could specify HTTP/1.1, HTTP2 [7], and SPDY [15]. As these are a list of arbitrary strings, unlike most other extensions, there is no standard set of possible application protocols.

GREASE (Generate Random Extensions And Sustain Extensibility) is a TLS mechanism intended to discourage middleboxes and server implementations from "rusting shut" due to ubiquitous static use of TLS extensibility points [8]. If clients only ever send a subset of TLS extension values, subpar (but still widely deployed) implementations may be tempted to hardcode those values or parse for them specifically. If this happens, later versions or implementations of TLS that attempt to include additional extensions may find that they cannot complete a TLS connection through buggy middlebox implementations. To discourage this, GREASE specifies that clients may send "random" extensions, cipher suites, and supported groups. Google has deployed GREASE in recent versions of Chrome, discouraging buggy server implementations that reject unknown extensions, cipher suites, or supported groups. Instead, such buggy implementations would be quickly discovered to not work with Google Chrome, prompting maintainers to fix the server or middlebox before it was widely deployed.

III. Measurement Architecture

Our institution's network consists of a full-duplex 10 Gbps link for the main campus network, including campus-wide WiFi traffic, lab computers, residence halls, and web services. In cooperation with our university's networking and IT support, we deployed a single 1U server with a dual-port Intel X710 10GbE SFP+ network adapter, with an Intel Xeon E5-2643 CPU and 128 GB of RAM. We received a "mirror" of the bi-directional campus traffic from an infrastructure switch. Of the packets that reach our server, we suffer a modest drop rate below 0.03%.

We used PF_RING to process packets from the NIC and load balance them across 4-cores (processes). We wrote our packet processing code in 1400 lines of Rust. We ignored packets that were not TCP port 443 or had incorrect TCP checksums, and kept an internal flow record of each new connection. Upon seeing a TCP SYN, we recorded the 4-tuple (source/destination IP/port) in a flow record table, and waited for the first TCP packet carrying data in the connection. We attempted to parse this data as a TLS Client Hello message, which succeeded 96.7% of the time. We note that this method will miss out-of-order or fragmented Client Hello messages.

A. Collected Data

In our study, we collected 3 kinds of information from the network, including counts and coarse grained timestamps of unique Client Hello messages, a sample of SNI and anonymized connection-specific metadata for each unique Client Hello, and Server Hello responses. We applied for and received IRB exemption for our collection, and worked with our institution's network and IT staff to ensure protection of user privacy.

Client Hellos For successfully parsed Client Hellos, we extracted the TLS record version, handshake version, list of cipher suites, list of supported compression methods, and list of extensions. When present, we extracted data from several specific extensions, including the server name indication, elliptic curves (supported groups), EC point formats, signature algorithms, and application layer protocol negotiation (ALPN). We then formed a fingerprint ID from this data, by taking the SHA1 hash of the TLS record version, handshake version, cipher suite list, compression method list, extension list, elliptic curve list, EC point format list, signature algorithm list, and ALPN list[4]. We truncated the SHA1 hash to 64-bits to allow it to fit a native type in our database. Assuming no adversarially-generated fingerprints, the probability of any collision (given by the birthday paradox) in a dataset of up to 1,000,000 unique fingerprints is below 10^{-7}. For each fingerprint, we recorded a count of the number of connections it appeared in for each hour in a PostgreSQL database.

Connection-specific information To provide more context for identifying fingerprints, we also recorded the destination server IP, server name indication (if present), and anonymized client IP /16 network from a small sample of connections, along with the corresponding Client Hello fingerprint. This sample data helps us determine the source implementation or purpose of a particular fingerprint.

Server Hellos In addition to Client Hello messages, we also collected the corresponding server hello in each connection, allowing us to see what cipher suite and extensions were negotiated successfully. For each Server Hello message, we parsed the TLS record version, handshake version, cipher suite, compression method, and list of extensions. We also included

[4] We specifically exclude server name from our fingerprint

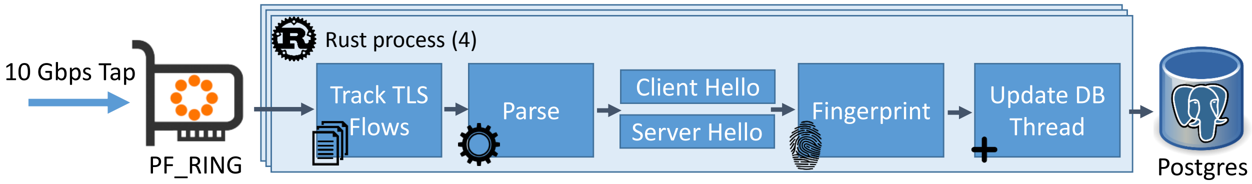

Fig. 2. **Collection Architecture** — We implemented our 10 Gbps collection infrastructure using PF_RING and 1400 lines of Rust, utilizing 4 processes. TLS client and ServerHello fingerprints and counts were aggregated and periodically written to a PostgreSQL database in a separate thread to avoid blocking the main packet parsing loop.

the data from the supported groups (elliptic curves), EC point format, and ALPN extensions.

Figure 2 shows a high level overview of our collection architecture.

BrowserStack In order to help link fingerprints to their implementations, we used BrowserStack—a cloud-based testing platform—to automatically conscript over 200 unique browser and platform combinations to visit our site, where we linked user agents to captured Client Hello fingerprints. Combined with normal web crawling bots and manual tool tests, our website collected over 270 unique fingerprints, with over 535 unique user agents.

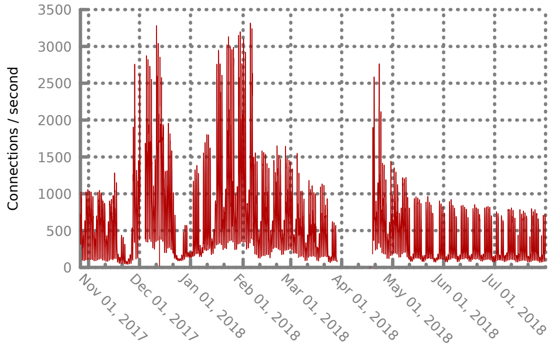

Fig. 3. **Connections observed** — We collected 9 months of TLS connections on our 10 Gbps campus network, observing over 11 billion connections. Noticeable is the diurnal pattern of traffic, as well as a decrease in traffic on weekends and holiday breaks.

B. Collection Details

Multiple Fingerprints Some TLS implementations generate several fingerprints. For example, Google Chrome generates at least 4 fingerprints, even from the same device. This is due to sending different combinations of extensions depending on the context and size of TLS request. Due to a server bug in the popular F5 Big IP load balancer, Client Hello messages between 256 and 511 bytes cause the device to incorrectly assume the message corresponds to an SSLv2 connection, interrupting the connection. When Google Chrome detects it would generate a Client Hello in this size range (for example by including a long TLS session ticket or server name value), it pads the Client Hello to 512 bytes using an additional `padding` TLS extension.

Browsers can also send different fingerprints from default due to end-user configurations or preferences. For example, Google Chrome users can disable cipher suites via a command line option [20].

We discuss clustering similar fingerprints in Section IV.

GREASE Because GREASE adds “random” extensions, cipher suites, and supported groups, implementations that support it would create dozens of unique fingerprints unless we normalize them. The specification provides 16 values that can be used as extension IDs, cipher suites, or supported groups, ranging from 0x0a0a to 0xfafa. While BoringSSL, used by Google Chrome, chooses these values randomly, we find their position is deterministic (e.g. first in the cipher suite list, and first and last in the extension list). We normalize these values in our dataset by replacing them with the single value 0x0a0a, which preserves the fact that an implementation supports GREASE while removing the specific random value that would otherwise generate unique fingerprints.

IV. High-level results

We collected TLS Client Hello fingerprints for approximately 9 months between late October 2017, and early August 2018 (ongoing). From December 28, 2017 onward, we additionally collected SNIs, destination IPs and anonymized source IPs from a sample of traffic, and we collected Server Hello messages starting on January 24, 2018.

Overall, we successfully collected and parsed over 11 billion TLS Client Hello messages. A small fraction (about 3.3%) of TLS connections failed to produce a parseable Client Hello, most commonly due to the first data received in a connection not parsing as a TLS record or Client Hello. This can happen for packets sent out of order or TCP fragments. We also ignored packets with incorrect TCP checksums, which happened with negligible frequency (0.00013%)[5].

Our collection suffered two major gaps, first starting on February 5 we only received partial traffic, and second between March 28 and April 19 we lost access to the tap due to a network failure. Figure 3 shows the number of Client Hello messages parsed over time, showing our gaps as well as the diurnal/weekend/holiday pattern of traffic.

漏資料是吧

Long tail fingerprint distribution

Our 11 billion TLS Client Hellos comprised over 230,000 unique fingerprints, a surprisingly high amount if we naively

[5] Due to a bug in the underlying `pnet` packet library, as many as 10% of packets were falsely reported to have incorrect checksums. We fixed this bug on January 25, 2018, but believe it had minimal impact on data-carrying TCP packets

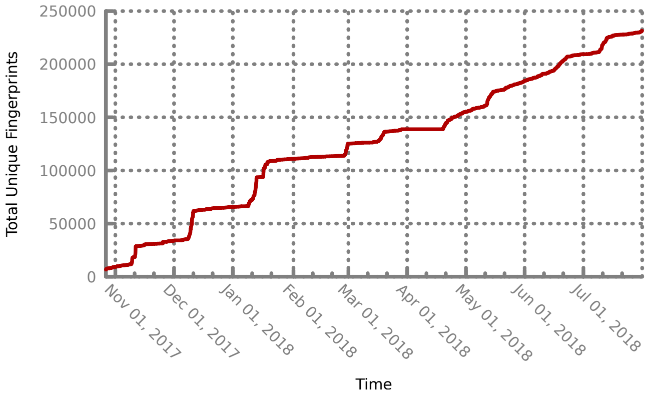

Fig. 4. **Total Unique Fingerprints** — The number of unique TLS ClientHello fingerprints observed rises slowly but steadily over time, reaching over 152,000 by May 2018. This rise is punctuated by short but rapid increases in the number of unique fingerprints, which we determined came from a small set of Internet scanners sending seemingly random ClientHello messages.

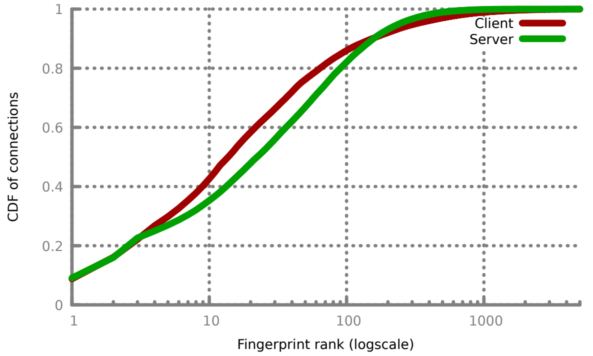

Fig. 5. **CDF of connections per fingerprint** — While we observed over 152,000 ClientHello fingerprints, most connections were comprised of a small number of fingerprints: over half the connections were of one of the top 12 fingerprints, and the top 3000 fingerprints make up 99.9% of connections observed. For servers, only 4,700 fingerprints were observed, with half of the connections using one of the top 19 server fingerprints.

August 2018

Rank	Client	% Connections
1	Chrome 65-68	16.51%
2	iOS 11/macOS 10.13 Safari	5.95%
3	MS Office 2016 (including Outlook)	5.34%
4	Chrome 65-68 (with padding)	4.62%
5	Edge 15-18, IE 11	4.05%
6	Firefox 59-61 (with padding)	3.62%
7	Safari 11.1 on Mac OS X	2.82%
8	iOS 10/macOS 10.12 Safari	2.49%
9	iOS 11/macOS 10.13 Safari (with padding)	2.42%
10	Firefox 59-61	2.22%

December 2018

Rank	Client	% Connections
1	Chrome 70 (with padding)	8.49%
2	iOS 12/macOS 10.14 Safari	7.55%
3	iOS 12/macOS 10.14 Safari (without ALPN)	4.15%
4	Chrome 70	4.10%
5	iOS 12/macOS 10.14 Safari (with padding)	4.09%
6	Edge 15-18, IE 11	3.27%
7	MS Office 2016 (including Outlook)	3.01%
8	iOS 10/macOS 10.12 Safari	2.72%
9	iOS 11/macOS 10.13 Safari	2.68%
10	Chrome 71 (with padding)	2.48%

TABLE I. **Top 10 Implementations.** — Most frequently seen fingerprints in our dataset and the implementations that generate them, for a week in August and December 2018. Despite being only 4 months apart, top 10 fingerprints changed substantially, as new browser releases quickly take the place of older versions.

assume each fingerprint corresponds to a unique implementation. Figure 4 shows the total number of unique fingerprints over time, rising from an initial 2145 to 230,000 over the course of several months.

Immediately visible are large steps in the number of unique fingerprints, signifying discrete events that produced a large number of fingerprints. We discover that these events correspond to a single monthly Internet scanner that produces very few connections, but appears to be sending a significant number of random unique fingerprints. In fact, over 206,000 of these fingerprints were only seen a single time, generally from the same /16 network subnet. While this scanner had a large impact on the number of unique fingerprints, its impact on connections was negligible. In the remainder of this paper, we report on the percent of connections that a fingerprint or fingerprints comprise, effectively allowing us to highlight common fingerprints without influence from this single scanner.

Figure 5 shows the CDF of connections seen per fingerprint for the most popular 5,000 fingerprints (for both Client and Server hello messages). 99.96% of all connections use one of top 5000 Client Hello fingerprints, and one of top 1310 Server Hello fingerprints.

Many of the top ten fingerprints (shown in Table I) are variants generated by the same TLS implementation: for example, Google Chrome 65 generates two fingerprints (with and without the padding extension), ranked 1st and 4th over the past week in our dataset. Chrome may also optionally include ChannelID extensions depending on if the connection is made directly or via an AJAX connection.

Fingerprint clusters

As mentioned, some implementations generate multiple TLS fingerprints, due to buggy middlebox workarounds, types of TLS connection, or user preferences. This raises the question of how many fingerprints does a typical implementation produce? We compared fingerprints by performing a basic Levenshtein distance over the components we extracted in the Client Hello. For example, two fingerprints that only differ by the presence of the `padding` extension would have a Levenshtein distance of 1, while a pair of fingerprints that differed by a dozen cipher suites would have a distance of 12.

To determine the prevalence of multiple fingerprint variants, we generated clusters of popular fingerprints. We looked at the 6629 fingerprints that were seen more than 1000 times in our dataset (accounting for 99.97% of all connections), and clustered them into groups if they were within a Levenshtein distance of 5 from another fingerprint in the group. This clustering resulted in 1625 groups, with the largest group having 338 fingerprints in it, corresponding to variants of Microsoft Exchange across several versions. Google Chrome 65 appeared in a cluster with 117 fingerprints, containing two weakly-connected sub-clusters. Half of this cluster represents fingerprints corresponding to an early TLS 1.3 draft, and the other half the current standard. Unsurprisingly, these fingerprints are long-tailed, with the top 10 fingerprints in the group responsible for 96% of the connections from the cluster.

Fingerprint churn

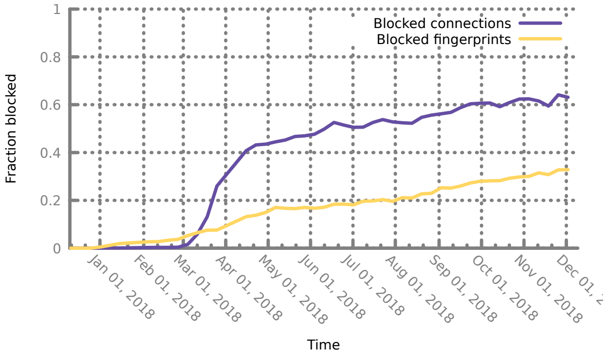

Fig. 6. **Fingerprint turnover** — Shows fraction of connections/fingerprints not seen during the first week. This roughly models the fraction that censor would overblock, if they took a static fingerprint snapshot and whitelisted it.

To measure how quickly fingerprints change and how this would impact a censor, we developed a simple heuristic. We compile a list of all fingerprints seen at least 10 times in the first week, and in subsequent weeks, compare fingerprints that were seen a substantial amount (10,000) times. This models the rate at which new fingerprints (with non-negligible use) are observed, and for a censor that employs a whitelist approach, describes the fraction of connections they would inadvertently block if they did not update their whitelist from an initial snapshot.

Figure 6 shows the increase in both fingerprints and connections blocked over time as new fingerprints are observed compared to an initial whitelist composed on the first week. In the steady state prior to March, the weekly average increase in blocked connections is approximately 0.03% (0.33% by fingerprints), suggesting that the rate of new fingerprints is steady but small. However, in March, both Google Chrome and iOS deployed updates that included new support for TLS features, creating a substantial increase in connections using new fingerprints. As a result, a non-adaptive whitelist censor would end up blocking over half of all connections after just 6 months.

Such large events could present a difficult situation for a whitelist censor, as new versions would be blocked until new rules were added. We conclude that whitelisting TLS implementations for a censor may be feasible, but requires a potentially expensive upkeep effort to avoid large collateral damage.

SNI

Subject Name Indication is a widely used TLS extension that lets the client specify the hostname they are accessing. Because it is sent in the clear in the Client Hello message, SNIs can be another feature that censors use to block. Many tools have different strategies for setting the SNI. Some use domain fronting, and set the SNI to a popular service inside the cloud provider (e.g. maps.google.com), while others choose to omit the SNI for ease of implementation or compatibility reasons.

As of August 2018, we observe only 1.41% of connections do not send the SNI extension, indicating that the circumvention strategy of omitting SNIs may potentially stand out and be easy to block. Indeed, the most popular SNI-less fingerprint in our dataset (accounting for 0.20% of all connections) sends a very unique cipher suite list not seen in any other fingerprints. This result impacts many circumvention tools, including Psiphon and Lantern that both produce Client Hello messages that do not include the SNI extension, suggesting that these connections may be easy for censors to block.

V. Censorship Circumvention Tools

Many censorship circumvention tools use TLS as an outer layer protocol. For example, domain fronting uses TLS connections to popular CDN providers with cleartext SNIs suggesting the user is connecting to popular domains hosted there [25]. Inside the TLS connection, the user specifies a proxy endpoint located at the CDN as the HTTP Host, relying on the CDN's TLS terminator and load balancer to route their traffic to the intended destination. As censors only see the unencrypted SNI, they cannot distinguish domain fronting connections from benign web traffic to these CDNs. Psiphon, meek and Signal use this technique to conceal their traffic from would-be censors [17], [32], [57]. However, mimicking connections in this manner is difficult in practice: any deviations from the behavior of true browsers that typically access these CDNs allows a censor to detect and block this style of proxy [30].

Other tools such as Psiphon and Lantern use TLS in a more natural way, connecting directly to endpoint proxies. These tools must also take care to mimic or otherwise hide their connections to look normal or face blocking.

Two different versions of meek were found to be detectable by Cyberoam [21] and Fortiguard [22] firewalls, which fingerprinted the Client Hello of meek and blocked it. At the time of incident with Cyberoam, meek was mimicking Firefox 38, which had over time dwindled in users, reducing overall collateral damage from blocking meek. Notably, censors were able to substantially reduce collateral damage by combining TLS fingerprinting with simple host blocking: blocking was limited to clients that both looked like Firefox 38 and tried to access specific domains used for fronting. Subsequently, meek started mimicking the newer Firefox 45, but was ultimately blocked by Fortiguard in the same manner when Firefox 45 became less popular.

Given the importance of Client Hello messages in identifying and potentially blocking censorship circumvention tools, we analyzed the fingerprints generated by several popular circumvention tools, and compared the relative popularity of those fingerprints in our dataset. Fingerprints that were seen much more frequently should in theory be more difficult for censors to block outright without collateral damage, while those that we rarely see in our dataset may be at risk of easy blocking. In this analysis, we assume that our dataset contains negligible traffic from these tools, as our institution is not in a censored region and users have little motivation to use them en mass.

We also provide a mechanism to easily test and analyze any application by submitting a pcap file to our website:

https://tlsfingerprint.io/pcap. Our website will list all the fingerprints extracted from the pcap file, and provide links with more details about their features, popularity, any user agents observed using these fingerprints, and similar fingerprints that can be compared.

A. *Signal*

Until recently, the Signal secure messaging application used domain fronting to circumvent censorship in Egypt, the United Arab Emirates, Oman, and Qatar [41]. Signal used Google App Engine as a front domain until April 2018, when Google disabled domain fronting on their infrastructure [39]. Signal switched to domain fronting via the Amazon-owned souq.com, but shortly after, Amazon disallowed domain fronting on their infrastructure, and notified Signal that it was in violation of its terms of service [38], [39]. Signal has since stopped using domain fronting, and direct access is now blocked in the above countries.

However, Signal still serves as one of the largest deployments of domain fronting, and we analyze both when it used Google and when it used Amazon.

When Signal was using Google for domain fronting, we analyzed both the iOS and Android versions of the application on real devices and collected the TLS fingerprints it generated when we signed up with phone numbers in the previously mentioned censored countries, triggering its domain fronting logic. For iOS, we found it generates the native iOS fingerprint, which appears in 2.14% of connections, making it the 11th most popular in our dataset at the time. It is unlikely a censor would be able to block Signal iOS from its Client Hello fingerprints alone.

However, on Android, the situation is drastically worse. Even on the same device, Signal Android generates up to four unique fingerprints when using domain fronting. Some of these fingerprints were never seen in our dataset, making it trivial for a censor to detect and block. Even the most popular fingerprint was seen in only 0.06% of connections, making it ranked 130th in popularity. It appears to be used by a small fraction of Android 7.0 clients that access googleapis.com. We confirmed these findings using two devices: a Google Pixel running Android 7.1 with Signal version 4.12.3, and a Samsung G900V running Android 6.0.1 with Signal version 4.11.5.

Signal on Android uses the okhttp[6] library to create TLS connections with three different "connection specs" that define the cipher suites and TLS version to be used in the Client Hello. Signal attempted to mimic 3 different clients for 3 fronts it used: Google Maps, Mail and Play. However, we identify two problems: First, the okhttp library disables certain cipher suites by default such as DHE and RC4 ciphers, which are specified in Signal's connection specs [12]. Although the library supports them, okhttp disallows their use without an explicit API call, which Signal did not use. Instead, the okhttp library silently drops these ciphers from the Client Hello it sends, causing it to diverge from the intended connection spec.

Second, even when the desired cipher suite list is unaltered, these cipher suites still correspond to unpopular clients, due to differences in other parts of Client Hello. For example, the `GMAIL_CONNECTION_SPEC`'s cipher suite list is most commonly sent by Android 6.0.1, and is seen in only 0.17% of connections. However, the Signal Client Hello is trivially distinguishable from the Android 6.0.1 fingerprint, as Signal does not specify support for HTTP/2 in its ALPN extension, while Android does.

By April 2018, Signal switched to using the Amazon-owned souq.com, and changed the TLS configuration to only use a single connection spec. This configuration still included 3 DHE cipher suites, which are ignored and not sent by the okhttp library. We also noticed that despite only the single spec, Signal generates two distinct TLS fingerprints, differing in which signature algorithms they list. Each of these fingerprints are seen in fewer than 100 times (<0.000001% of connections).

As of May 2018, Signal no longer uses domain fronting, making these issues moot. Nevertheless, these examples illustrate several challenges in mimicking popular TLS implementations. First, libraries are not currently purpose-built for the task of mimicry, emphasizing security over an application's choice of cipher suites. Second, in addition to cipher suites, an application also must specify extensions and their values in order to properly mimic other implementations.

B. *meek*

meek is a domain fronting-based pluggable transport used by Tor to circumvent censorship [14], [16], [17]. meek tunnels through the Firefox ESR (Extended Support Release) version that is bundled with the Tor Browser. Unfortunately, the Client Hello of Firefox and Firefox ESR may diverge, which eventually makes the fingerprint of ESR versions relatively uncommon, allowing meek to be blocked with smaller collateral damage. As of August 2018, meek uses Firefox 52 ESR whose corresponding Client Hello is the 42nd most popular fingerprint in our dataset, seen in approximately 0.50% of connections. The majority of regular Firefox users have migrated to Firefox versions 59+, whose most popular fingerprint is ranked 6th in our dataset seen in 3.64% of connections.

Thus, meek is using a version of Firefox that is once again approaching obsolescence, potentially allowing censors to block it without blocking many normal users. Unfortunately, meek must wait for updates to the underlying Tor Browser to receive updated TLS features, making it relatively inflexible in its current design.

C. *Snowflake*

Snowflake [18] is a pluggable transport for Tor that relies on a large number of short-lived volunteer proxies, similar to Flashproxy [24]. Clients connect to a broker via domain fronting, and request information about volunteers running browser proxies, and then connect to and uses those proxies over the WebRTC protocol.

Snowflake is under active development, and its authors were aware of potential TLS fingerprintability issues. Indeed, we find that Snowflake (built from git master branch on April 17, 2018) generates a fingerprint that is close to, but not exactly the same as the default Golang TLS fingerprint. In particular,

[6] https://square.github.io/okhttp/

it diverges by including the NPN and ALPN extensions, and offers a different set of signature algorithms. As a result, this fingerprint is seen in fewer than 0.0008% of connections, making it susceptible to blocking.

D. Outline

Outline is a private self-hosted proxy based on Shadowsocks, a randomized protocol that attempts to look like nothing. Outline provides a GUI interface that guides the user through the Shadowsocks setup process, including purchase of a VM on DigitalOcean. During the purchase, the installation script uses TLS, leveraging the Electron framework (based on Chromium). As of our tests in May 2018, we find that Outline's TLS fingerprint matches that of Chrome version 58-64.

While this fingerprint is decreasing in popularity with the release of Chrome 65, it still remains common: We observed it in 2.10% connections in the first week of May (vs. 8.76% of connections over our full dataset). As of August 2018, the weekly rate of connections with this fingerprint has fallen to 1.72%. Still, the Outline installation process is unlikely to be blocked based on the generated TLS fingerprint in the short term, though it must take care to update before use of this fingerprint wanes further.

We did not evaluate the rest of the communication protocols used by Outline after installation, as it does not use TLS.

E. TapDance

TapDance [65] uses refraction networking to circumvent censors, placing proxies at Internet service providers (ISPs) outside censoring countries, and having clients communicate with them by establishing a TLS connection to a reachable site that passes the ISP. As it connects to innocuous websites, the TapDance client must make its TLS connection appear normal to the censor while it covertly communicates with the on-path ISP.

As of May 2018, TapDance uses a randomized Client Hello, which protects it against straightforward blacklisting. However, the randomized fingerprints generated by TapDance are not found in our dataset of real-world fingerprints, which could allow a censor to block it by distinguishing randomized Client Hellos from typical ones, or by simply employing a whitelist of allowed TLS Client Hello messages.

F. Lantern

Lantern uses several protocols to circumvent censorship, mainly relying on the randomized Lampshade pluggable transport [43]. However, as of February 2018, several parts of Lantern still use TLS as a transport, allowing us (and censors) to capture its fingerprint.

We observed that Lantern uses a Golang TLS variant that sends a Session Ticket extension, and doesn't send the server name extension. This variant does appear in our dataset, however, at a very low rate: approximately 0.0003% of connections, ranked 1867 in terms of popularity.

Lantern uses the Session Ticket to communicate information covertly out-of-band to their server. However, this use makes their fingerprint differ from the default Golang TLS, illustrating that application-level demands may result in observably different handshakes, ultimately allowing a censor to block them.

G. Psiphon

Similar to Lantern, Psiphon also uses several circumvention transports, including domain fronting over TLS. We obtained a version of the Psiphon Android application that only connects using TLS from the app's developers, allowing us to collect TLS fingerprints generated by it. Psiphon cycles through different Chrome and Android fingerprints until it finds an unblocked one that allows them to connect to their servers. Such a diverse set of fingerprints may help evade censorship, even if most fingerprints get blocked. However, a censor with a stateful DPI capability may also be able to use this feature to detect (and ultimately block) Psiphon users.

We find Psiphon successfully mimics Chrome 58-64, generating two fingerprints ranked in the top 20 in our dataset, but is less successful at mimicking legacy Android: fingerprints supposedly targeting Android were seen in fewer than 50 connections out of the 11 billion. We also determined that Psiphon sometimes mimics Chrome without an SNI to evade SNI-based blocking, generating a blockable fingerprint seen in fewer than 0.0002% of connections.

H. VPNs

We analyzed OpenVPN and 3 services that advertised an ability to circumvent censorship: IVPN, NordVPN and privateinternetaccess.com. VPNs tend to use UDP by default for performance benefits, which we did not collect in our measurement system. However, we extracted the cipher suites and extensions from OpenVPN's UDP TLS handshake, and found that the combination is rare in our (TCP-based) dataset. This may suggest that OpenVPN has a distinctive fingerprint, given the unique set of features in its fingerprint (85 cipher suites, 13 groups, 14 signature algorithms, and rare set of extensions), but we cannot be sure without collecting UDP TLS connections.

We did recover one TCP TLS fingerprint from NordVPN, which is a circumvention plugin in Google Chrome. However, this plugin uses the API of the host browser to make TLS requests, making it indistinguishable from the version of Google Chrome it is installed in.

I. Vendor Response

We notified the authors of the censorship circumvention tools about respective fingerprintability issues, and provided additional data about fingerprints as well as potential defenses.

In response to our disclosure, developers of Psiphon, Lantern, and TapDance integrated our uTLS library described in Section VII to take advantage of alternative mimicry options and have greater control over TLS features. The Snowflake and meek authors were aware that the current fingerprint was not ideal, but didn't have immediate plans to fix it. Snowflake is still in active development; and meek will keep tunneling through Firefox ESR in short term, due to the effort involved in changing it, but is considering going back to mimicking, rather than tunneling. We disclosed our findings to Signal via

Tool	Version/Date	Rank [all time]	% Connections
Psiphon	Jan 2018	1	8.76%
		9	2.42%
		62	0.25%
		198	0.04%
		203	0.04%
		500	0.01%
		2190	0.0002%
		14397	0.0000%
		16814	0.0000%
Outline	May 2018	1	8.76%
meek	TBB 7.5	42	0.50%
Snowflake	April 2018	1378	0.0008%
Lantern	4.6.13	1867	0.0003%
TapDance	May 2018	random	-
Signal	4.19.3	11468	0.0000%
		12982	0.0000%

TABLE II. **Tool Fingerprintability** — Summary of all TLS fingerprints generated by censorship circumvention tools and their rank and percent of connections seen in our dataset as of early August 2018. Highlighted in red are fingerprints seen in relatively few ($< 0.1\%$) connections, putting them at risk of blocking by a censor.

email and a GitHub issue prior to their removal of domain fronting, but did not receive any response besides the GitHub issue being silently deleted.

VI. Defenses & Lessons

While the TLS protocol provides plenty of cover traffic for circumvention tools, there are many challenging details that tools must get right in order to successfully evade censorship. There are two high-level detection evasion strategies that circumvention tools employ when choosing protocols [59]: first, they may try to *mimic* one or more existing implementations, making it difficult to distinguish them and increasing the collateral damage to blocking connections that look like the tool. Second, tools may try to generate *random* protocol features, to prevent censors from being able to identify and block the tool with a blacklist approach. In this section, we investigate how these techniques can be applied to TLS implementations in censorship circumvention tools.

Mimicking Clients that choose to mimic other TLS implementations face several challenges. First, tools must *identify a popular implementation* to mimic, which is typically done by choosing a popular web browser. However, it may also be done by choosing a large number of individually less-popular but collectively popular implementations, and mimicking among them. Second, the tool must *support* the cipher suites, extensions, and features present in the popular implementation(s). For instance, if a tool mimics Chrome and sends but does not actually support a CHACHA20 cipher suite, the server may select that cipher suite for the connection, causing the tool to abort the connection. Not only does this cause compatibility issues, it gives an observant censor a way to identify users of a tool.

We note that this problem can be partially mitigated if the server is controlled by the tool maintainer, as they can choose to select only cipher suites and extensions that they know their tool to support. However, this is not the case in tools that do not control both endpoints, such as domain fronting, refraction networking[7], and tools that generate cover traffic to other servers. In addition, intricacies of the circumvention protocol may limit the features that a tool can use, even if implemented. For example, some domain fronting tools cannot send the SNI extension to certain CDNs.

Finally, the tool must *maintain* support, as the popular implementations they mimic change over time, as do the features they support due to automated patches and updates. For example, although meek has been successful at mimicking multiple versions of Firefox, it has lagged behind the Sisyphean task of keeping pace with updates to the Firefox TLS implementation.

Randomizing Tools that randomize their generated TLS Client Hello messages have the advantage that they do not have to identify or track support for popular implementations. However, this strategy can only work if there is a sufficient number of similar-looking connections that prevent the censor from distinguishing it. For instance, Figure 6 demonstrates the rate at which new connections would hamper a censor's ability to use a whitelist. Censors could also distinguish randomized Client Hello messages by capturing distributions or other heuristics that are not properly mimicked by a tool's randomized fingerprint. For example, if the tool naively picks from a set of supported extensions, a censor may notice that no other implementation supports a particular pair of extensions simultaneously. When the circumvention tool randomly selects both extensions in this pair, the censor can identify and block the user. Thus, the random fingerprint strategy must carefully mimic the *distribution* of the global TLS implementation population.

We note that all tools must implement these features either by creating their own libraries or using existing ones that are generally ill-suited to the task of fine-grained control over the TLS handshake. This challenge is illustrated by Signal's use of the okhttp library, which silently removed cipher suites that Signal specified. Other TLS libraries may ignore supported cipher suite order, making it difficult for applications to produce specific TLS handshakes. In the following section, we describe our library that is purpose-built for providing applications control over their TLS connection.

VII. uTLS

TLS fingerprinting remains a looming threat for anti-censorship tools, and as we have shown, even tools that attempt to defend against it can often fall short. Indeed, mimicking is hard to get right: there are lots of features to keep track of and implement, the mimicked fingerprint could rapidly go obsolete, or the tool's underlying library could silently change the fingerprint.

There may also be unexpected or complicated dependencies that prevents simply parroting Client Hello messages seen. For example, GREASE values generated by Google Chrome used to be deterministic and depend on the value of the Client Random, but this was changed in favor of random values. In addition, cipher suites can influence other parts of the header (or server response), such as by having a special meaning (SCSV cipher suites), by defining what TLS version is used, or triggering the inclusion of an extension. Finally, extensions may affect each other, for example, the presence and size of padding extension can depend on the size of the Client Hello.

[7] formerly decoy routing

An implementation that failed to mimic these subtleties could be identified by a censor.

To assist censorship circumvention tools, we created a TLS library[8] that aims to protect against TLS fingerprinting and (among other features) allows developers to easily mimic arbitrary Client Hello messages. We develop our library as a fork of Golang's native TLS library `crypto/tls`, adding over 2200 new lines of code.

As of August 2018, three circumvention tools have adopted uTLS library: Psiphon, Lantern, and TapDance all use uTLS to allow them greater control over TLS features, and make it easier to mimic popular implementations.

A. Design

uTLS is designed to be an addition to the standard `crypto/tls` library and minimizes changes to core Go files, enabling us to easily auto-merge from upstream. This allows us to keep uTLS up-to-date with the underlying standard library, and adopt any new features and bug fixes that come in the future. In addition, this allows us to rely on the performance and security of `crypto/tls`: uTLS simply fills the Client Hello and leaves execution of the TLS handshake up to the standard functionality.

Our choice of Golang as the language for uTLS is motivated by several reasons. First, it is a popular language used in several censorship circumvention tools, including Lantern, meek, Psiphon, Snowflake, and TapDance, allowing easier integration. For tools that are not written in Go, integration should still be possible via Go's language bindings [35]. Second, Golang is memory safe (bounds checked), decreasing our worry of introducing control flow vulnerabilities into tools that integrate uTLS, despite containing network serialization code.

a) Low-level access to the handshake: uTLS provides write access to any fields of the Client Hello message, allowing implementations to specify their own cipher suites, compression methods, client random, extensions, etc. Developers can compose a Client Hello using uTLS structures, or by manually specifying the bytes of a raw Client Hello for uTLS to use. In addition, uTLS provides structured read access to handshake state, including the Server Hello, Master Secret, and key stream.

b) Mimicry: Users can also select from a set of preset built-in Client Hello messages. As of August 2018, uTLS includes defaults for Chrome 64, Firefox 58, and iOS 11, and we plan to add support for new versions of browsers, operating systems, and other popular devices. Mimicking could be hard to get right, but we verified uTLS' ability to mimic popular clients by comparing the fingerprints it generated to those in our dataset.

We note that when uTLS mimics a Client Hello of another implementation, it may potentially advertise support for features it does not actually implement. For instance, it may send a cipher suite that, if selected, uTLS will be unable to use. We measure this risk in Section VII-B.

c) Randomized fingerprints: Given the long tail distribution of Client Hello fingerprints in our dataset, uTLS also supports generating random fingerprints. Although these are unlikely to correspond to popular implementations and fingerprints seen, censors may have a hard time constructing a comprehensive whitelist of TLS fingerprints, making it difficult to block random ones. Similar techniques have been used by other randomized protocols for censorship circumvention, such as obfsproxy [5] and ScrambleSuit [64], which attempt to look like no protocol at all. Our random fingerprints extends this idea at the TLS layer, ensuring that packets are valid TLS messages, but making it difficult for censors to blacklist the specific implementation.

d) Using multiple fingerprints: Mimicking multiple fingerprints makes it possible for a circumvention tool to operate even when a subset of its fingerprints are blocked. To support this, uTLS can optionally cycle through a popular set of fingerprints in its handshakes until an unblocked working one is found. Thus, if uTLS is able to properly mimic even one implementation, it will be more difficult for a censor to block this strategy. uTLS automatically retries the latest working fingerprint when reconnecting to minimize unnecessary changes to its fingerprint. An example usage may be found in Appendix, Listing 1.

Automatic code generation While uTLS makes it easy to manually specify parts of the Client Hello, we provide an additional feature to make this even easier. Our website produces automatically-generated code for each fingerprint in our dataset, allowing developers to simply copy and paste to configure uTLS to mimic a given fingerprint. This feature allows developers to easily keep their tools up-to-date, and could even allow this to potentially be fully-automated: continuous integration scripts could watch for changes in fingerprint popularity, and either alert developers or possibly automatically pull in new code to use more recent fingerprints.

We note that automated fingerprint code may generate Client Hellos that uTLS does not yet fully support, potentially causing the handshake to fail if the server supports those features. We explicitly identify and warn the developer when this is the case, as our automated code tracks the features uTLS supports.

Fake Session Tickets uTLS provides the ability to send arbitrary session tickets, including fake ones. This is useful when the tool developer also controls the server, which must accept the fake session ticket or generate a real one for further out-of-band distribution. This technique allows servers to save a round trip time and avoid sending a Certificate or Key Exchange message, giving the censor fewer messages and information to block on. To support mimicking, we track commonly used Session Ticket sizes on our website[9].

B. Measuring feature support

To enable mimicking other TLS implementations, uTLS allows the developer to advertise support for TLS extensions and cipher suites that are not actually supported by our library. If the server does not select or use these cipher suites or extensions, the connection will function normally. However,

[8] https://github.com/refraction-networking/utls

[9] https://tlsfingerprint.io/session-tickets

if the server selects a cipher suite that is not implemented by uTLS, the connection will visibly break. These risks do not impact tools that make connections to servers under the developers' control, as those unsupported features can be easily disabled server side.

We measure how many fingerprints in our dataset uTLS can support without this risk. We classify a fingerprint as "fully supported" if uTLS is able to handle every feature in it (e.g. the server could select any cipher suite, curve, or extension, and the connection would succeed). We classify a second group of "optionally supported" fingerprints that are also supported by uTLS but may include weaker ciphers, such as such as `TLS_RSA_WITH_AES_256_CBC_SHA256`, that were disabled by the underlying Golang library. uTLS users may choose whether to enable those weaker ciphers, or to let the connections fail if the weaker ciphers are chosen by a server.

Note that a fingerprint not being fully supported doesn't always lead to unsupported feature getting chosen by server: it might be a low-priority, or not supported by the server either. Weaker ciphers are only likely to be picked in the wild if the client communicates with an outdated server.

As of August 2018, uTLS fully supports 21940 fingerprints (9.3%), which were seen in 5.9% of connections collectively. The top ranked fully supported fingerprint is the 9th (all-time) most popular fingerprint in our dataset, which is a fingerprint generated by Chrome 61-64. If weaker "optionally supported" CBC ciphers are allowed, then uTLS supports 22616 fingerprints (9.6%), which were seen in 37.3% of collective connections. This includes 30 fingerprints in the top 100, including the 3rd most popular fingerprint (generated by Outlook 2016). As mentioned, uTLS code for using all of these fingerprints can be automatically generated by our website, requiring minimal effort from the developer.

We also use our data to learn which additional features would give uTLS the most additional coverage in terms of supported fingerprints, to know which features we might want to focus on adding to the library next. As of August 2018, supporting the ChannelID extension alone will allow us to fully support 245 more fingerprints which would account for an additional 20% of connections seen. We note that these fingerprints could be mimicked in uTLS now, as the Channel ID is unlikely to be supported by servers: only 2.9% of Server Hello messages used the extension. This extension aims to secure connections by cryptographically authenticating client to the server, allowing to bind cookies and tokens to particular client's channel, which has to be explicitly implemented and integrated with the application layer on the server.

C. TLS 1.3

As of March 2018, TLS 1.3 [45] has been standardized and is being rolled out in major browsers. TLS 1.3 offers several advantages over previous versions, including decreased network round trips in new connections (improving performance), and encrypted handshakes (improving privacy).

Our motivation to support TLS 1.3 in uTLS is twofold: first, several features such as encrypted certificates and encrypted SNI[10] (ESNI) may prove useful for evading censors [23]. Second, as popular browsers begin to implement and send TLS 1.3 handshakes, circumvention tools will soon want to mimic them to continue blending in with popular implementations.

Interestingly, TLS 1.3 Client Hellos look similar to those in TLS 1.2: in fact, TLS 1.3 still sends a handshake version corresponding to TLS 1.2 to allow implementations to work in the presence of buggy middleboxes and servers that cannot handle other values. TLS 1.3 instead adds functionality via several new extensions in the Client and Server Hello messages.

Although our fingerprints already include the presence and order of all extensions in a given Client Hello, we only parse and include a handful of extensions' data in our fingerprint. This means if many implementations send the same set of extensions but include different data, we would mistakenly classify them as the same fingerprint. This is particularly a concern for TLS 1.3 handshakes, which heavily rely on new data-carrying extensions. To address this, we reviewed the new extensions in TLS 1.3, and added the body data of popular extensions that do not change per-connection. For example, `supported_versions` contains a list of supported TLS versions ordered by preference. As different implementations can announce different versions they support, we add this data to our fingerprint. So far, we have observed 40 distinct values of this extension announcing support for the various TLS versions and drafts. Table III shows the most popular extensions we have collected as of December 2018, and highlights the ones whose data we include in our fingerprint.

Extension	Conns	Extension	Conns
supported_groups	99.4%	GREASE	30.2%
server_name	99.3%	**psk_key_exchange_modes***	28.7%
signature_algorithms	97.8%	**supported_versions***	28.7%
ec_point_formats	96.9%	**key_share***	28.6%
extended_master_secret	86.8%	NPN	27.3%
status_request	85.7%	**compressed_certificate***	24.8%
renegotiation_info	81.9%	ChannelID	20.5%
ALPN	71.9%	heartbeat	5.0%
signed_certificate_timestamp	66.9%	token_binding	3.9%
SessionTicket	56.0%	pre_shared_key*	3.1%
padding	32.3%	**record_size_limit**	2.5%

TABLE III. **TOP EXTENSIONS** — WHILE WE INCLUDE THE PRESENCE AND ORDER OF ALL EXTENSIONS IN OUR FINGERPRINT, **BOLD** DENOTES EXTENSIONS WHOSE DATA WE ADDITIONALLY PARSE AND INCLUDE IN OUR FINGERPRINT; * MARKS EXTENSIONS NEW IN TLS 1.3.

As uTLS is built on Golang's crypto/tls library, we were able to merge its TLS 1.3 support into uTLS, allowing us to mimic Firefox 63 and Chrome 70, which both send TLS 1.3 handshakes. With some additional implementation work to support new extensions, we expect to be able to fully support over 8% of all TLS connections automatically (up from 5% currently), and optionally support over 37% if we enable weak ciphers.

VIII. OTHER RESULTS

In this section, we present on other findings from our TLS dataset that are relevant to censorship circumvention tools.

[10] Not yet standardized

A. Server Hello Analysis

As of August 2018, we collected approximately 5,400 unique Server Hello fingerprints, substantially fewer than the number of unique Client Hello fingerprints. This is in part due to Server Hello messages having less content, as it specifies only a single cipher suite and compression method, rather than the full list that the server supports. On the other hand, while a client implementation might generate a single or small collection of Client Hello fingerprint, servers can potentially generate distinct fingerprints in response to different Client Hello messages. For example, the single most popular external IP address (corresponding to Google) sent 199 unique server fingerprints from 1494 Client Hello fingerprints sent to it. Looking at the responses to only the most popular Client Hello message, there are 750 different Server Hello fingerprints, suggesting that the actual number of distinct TLS server implementations and configurations that these clients talk to may be close to this value.

Selected Ciphers

Using our collected information on Server Hello messages, we can compare the set of offered cipher suites by clients, and discover what cipher suites are actually selected and used in practice by servers. This is useful for circumvention tools as it provides evidence of many unselected cipher suites that clients can offer without having to actually support.

Excluding the long tail of fingerprints seen only once, in our Client Hello fingerprints, there were over 7900 unique sets of unique cipher suites. These sets enumerate 522 cipher suite values, which is greater than the number of standardized cipher suites, for reasons described in Section VIII-B.

Analyzing the unique cipher suites that are selected by servers, however, we find just 70 cipher suites ever selected, with the top 10 accounting for over 93% of all connections. Interestingly, the most popular cipher suite across all Client Hellos (TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA) is only selected in approximately 1% of connections. This shows there are many cipher suites that servers will rarely or never choose, and circumvention tools are free to present them in their Client Hello messages without having to support those cipher suites.

B. Non-standard parameters

Our collection tool ignores malformed Client Hellos that cannot be parsed, but even well-formed Client Hello messages may still contain invalid parameters. For example, of the 65536 possible values for 2-byte cipher suites, only 338 values are recognized and standardized by the Internet Assigned Numbers Authority (IANA) [46], with the remainder either unassigned or reserved for private use. Similarly, only 28 values are defined for the 2-byte extension field. We note that TLS 1.3 proposes values for an additional 5 cipher suites and 10 extensions, which we include in our analysis.

This allows us to locate TLS Client Hello fingerprints that specify non-standard extensions or cipher suites. In total, over 138,000 fingerprints accounting for 13.14% of connections contained at least one non-standardized cipher suite or extension value. The commonality of support for non-standard features suggests it may be difficult for a censor to fully understand or whitelist all commonly-used fingerprints, as many do not strictly conform to the standard. Table IV shows the breakdown of non-standard parameters.

	Fingerprints	% Connections
TLS 1.3 draft ciphers	1002	10.626%
Legacy ciphers	82992	1.392%
GOST ciphers	95548	0.051%
Outdated SSL ciphers	106439	0.097%
Unknown ciphers	137999	0.039%
Total non-standard ciphers	**143060**	**12.106%**
TLS 1.3 draft extensions	715	10.626%
Legacy Extensions	441	0.154%
Extended Random	340	1.445%
Unknown extensions	367	0.899%
Total non-standard extensions	**1404**	**11.677%**

TABLE IV. **Non-standard parameters** — Breakdown of the number of unique Client Hellos (fingerprints) and the share of connections they appear in that send non-standard cipher suites or extensions. While TLS 1.3 draft ciphers and extensions are the majority, we still find unknown ciphers and extensions in use.

	Fingerprints	% Connections
DES	191459	0.90%
3DES	236859	67.0%
EXPORT	194418	0.66%
RC4	223900	8.19%
MD5 (Cipher)	200608	7.15%
MD5 (Sigalg)	4385	0.74%
SHA1 (Sigalg)	114615	97.6%
TLS_FALLBACK	787	0.03%

TABLE V. **Weak ciphers** — We analyzed the percent of connections offering known weak cipher suites. We also include TLS_FALLBACK_SCSV, which indicates a client that is falling back to an earlier version than it supports due to the server not supporting the same version.

C. Weak Ciphers

We observe a small fraction of clients continue to offer weak or known-broken ciphers, including DES [27], [58], Triple-DES (3DES) [10], and RC4 [2], [61]. More concerning, we still see clients supporting export-grade encryption, which negotiates intentionally weakened keys and has been recently found to enable modern vulnerabilities [9], [6].

TLS can also employ hash functions with known collisions, such as MD5 [55], [52] and SHA1 [63], [56]. While collisions may not enable attacks when used in the HMAC construction employed by TLS cipher suites, they can introduce problems when used in signature algorithms, as collisions there can allow an attacker to forge CA permissions [52]. This means that MD5 and SHA1 may not be problematic as cipher suites, but are when offered as a signature algorithm. We present both uses for completeness.

Clients can also signal that they have fallen back to a lower version of TLS by sending the TLS_FALLBACK_SCSV cipher suite [40]. While its presence does not indicate a weakness in a client, it does indicate a suboptimal mismatch between client and server versions.

Table V summarizes the fingerprints and percent of connections we observed clients offering these weak cipher suites and signature algorithms. While circumvention tools would likely avoid using such weak cipher suites (lest it enable a censor to successfully break their TLS connections), this further demonstrates the wide range of TLS implementations present on the modern Internet, once again showing how long legacy code can remain in use.

IX. Related Work

A. Passive TLS Measurements

Several studies have measured TLS (and SSL) by passively observing traffic as we have in our study. However, the vast majority of these studies focus mainly on certificates and the Certificate Authority ecosystem. For example, in 2011 Holz et al. analyzed 250 million TLS/SSL connections and extracted certificates in order to study the existing landscape of certificate validation [29]. In addition to uncovering the "sorry state" of the X.509 certificate PKI, they briefly analyzed selected cipher suites, finding that RC4-128, AES-128, and AES-256 were the most popular cipher suites used at the time, with TLS_RSA_WITH_RC4_128_MD5 selected in between 20 and 30% of connections. Today, 7 years later, we find that same cipher is selected in only 0.001% of connections, and offered by clients in only 8.4% of connections. Later, Holz et al. studied the use of TLS in email clients [28]. Lee et al. performed active scans of a sample of TLS/SSL servers in order to study ciphers supported and certificates in 2007 [36]. In 2012 the *SSL Notary* studied TLS/SSL certificates collected from live traffic [3]. The SSL Notary continues to run today[11], though still mainly focused on certificates and servers rather than clients. In 2014, Huang et al. described a way to detect forged SSL certificates via a flash plugin that could observe the raw certificate sent to the user [31].

B. Client Hello Fingerprinting

Several studies have used Client Hello messages to fingerprint TLS implementations. Most notably, in 2009, Ristić described how to fingerprint SSL/TLS clients by analyzing parameters in the handshake, including the cipher suites and extensions list [47], [34], [48]. Several works have since observed that these fingerprints can be used to identify and fingerprint third-party library use in Android applications [44], and detect malware [11], [4]. Durumeric et al. used TLS fingerprints and compared them to browser-provided user agents on a popular website to detect HTTPS interception by antiviruses and middleboxes [19].

While these works used Client Hello message to identify clients, we analyze the distribution of clients, ciphers and tls versions used, and fingerprintability of censorship circumvention tools, which to our knowledge has not been studied in this context.

C. Traffic Obfuscation Analysis

There are 2 general techniques [59] that censorship circumvention tools employ to avoid identification: mimic allowed type of content, or randomize the traffic shape to prevent blacklisting. In the former case, developers would have to eliminate all disparities between circumvention circumvention tool and imitated protocol, and will protect against whitelisting, as long as mimicked application is popular or important enough to avoid blocking. Houmansadr et al. demonstrated [30] that it is very difficult to successfully mimic application layer application. Randomized protocols, such as obfs4, while not being able to defend against whitelist approach, may counter blacklisting, which is used more commonly. Study by Liang Wang et al. examined attacks based on semantics, entropy, timing and packet headers [62], and demonstrated efficiency of entropy-based classifier in detecting obfs3 and obfs4. In 2013, tunneling loss-intolerant protocols over loss-tolerant cover channels, was shown[26] to allow censors to interfere with the channel safely, without disrupting intended use of cover-protocol. For details on observability and traffic analysis resistance of existing anti-censorship tools, reader can refer to Systematizations of Knowledge[33], [59].

lib•erate [37] is a library, that takes alternative approach, and instead of hiding the traffic, it features numerous techniques that use bugs in DPI to evade identification. Even though all proposed evasion techniques are susceptible to countermeasures, it might be cheaper for anti-censorship community to integrate and update lib•erate, than for censors to fix all the bugs in DPI boxes.

X. Discussion

A. Ethical Considerations

Studying real-world Internet traffic requires care to ensure user privacy. We designed our collection infrastructure to anonymize or discard potentially sensitive data. For example, we collected only the /16 subnet of the source IP address and SNI value for each connection. This allows us to tell if a connection originated on our campus, but not what individual user generated it. For connections originated externally, we often cannot even determine what AS the source was located in.

We applied for and received IRB exemption for our collection methodology, and we worked closely with our institution's IT and networking staff, who approved the specifics of our collection methods. We have responsibly disclosed our findings regarding the observability of the censorship circumvention tools, and are continuing to work with their respective developers to discuss and offer potential solutions.

B. Dataset Limitations

Although we have captured billions of TLS connections, there are limitations to what our infrastructure can collect. For example, fragmented TLS messages and out-of-order TCP packets are not parsed by our system. In addition, because we received the full-duplex 10 Gbps mirror of campus traffic on a single (half-duplex) port, it is possible for our copy of network traffic to saturate when the combined bi-directional traffic exceeds 10 Gbps. This occurs for several hours each day during peak load. We performed a simple experiment to quantify how this impacts our collection of TLS fingerprints.

Every hour, we made 100 TLS connections through our campus with a unique TLS fingerprint that did not appear in our dataset. This allowed us to see at what time of day we dropped fingerprints: any hour where we received fewer than 100 of these fingerprints indicated data loss. Over 88% of the hours we ran our experiment recorded all 100 of our test connections. However, during peak hours, which lasted approximately 5 hours per weekday, the minimum number captured

[11] https://notary.icsi.berkeley.edu/

in an hour was 43, and the median was 80. We conclude that the only times we do not capture TLS fingerprints is when the tap switch cannot forward us packets due to congestion, and all other times we receive and properly record practically all connections.

C. Future Work

Client Hello messages provide a rich amount of features useful in fingerprinting TLS implementations, but there are other messages in the TLS connection that could be used to detect or block tools. For instance, once the connection is established and sends encrypted records, the lengths of these encrypted records may reveal differences between implementations [62]. Collecting and better understanding the distribution of these (in conjunction with the information gleaned from Client Hello messages) could greatly help circumvention tools be more robust.

Another direction could be to study *user behavior* to better understand if existing tools that pretend to be users visiting popular CDNs or websites (e.g. in domain fronting or refraction networking) are easily distinguishable by the pattern or timing of connections they make.

Beyond TLS over TCP, measuring UDP TLS may be useful in performing similar analysis on DTLS protocols, such as those used by the VPN tools we investigated.

XI. Conclusion

We have analyzed real-world TLS traffic in the context of censorship circumvention tools, focusing on the first protocol messages sent between clients and servers that may allow censors to identify tools and implementations. We analyzed several circumvention tools that use TLS in various ways, and find problems with nearly all of them. To address these systemic problems, we have developed the uTLS library, designed to generated arbitrary Client Hello messages and provide applications full control over the TLS handshake, enabling them evade identification and blocking with minimal developer effort. We release our collected data, combined with tools to facilitate further analysis at https://tlsfingerprint.io.

Acknowledgements

We would like to thank the University of Colorado IT and Network Security team, particularly Dan Jones and Conan Moore for their valuable assistance and feedback in setting up our tap infrastructure. We thank J. Alex Halderman for his feedback in early drafts of this work. We also thank the developers at the circumvention projects we contacted, including Rod Hynes from Psiphon, and Ox Cart from Lantern, and David Fifield for their discussion on uTLS. Finally, we are deeply grateful to Alphabet Jigsaw, particularly Ben Schwartz, who helped to design and implement uTLS at the initial stages of the project and continued discussion thereafter.

References

[1] Sadia Afroz and David Fifield. Timeline of Tor censorship. http://www1.icsi.berkeley.edu/~sadia/tor_timeline.pdf.

[2] Nadhem J AlFardan, Daniel J Bernstein, Kenneth G Paterson, Bertram Poettering, and Jacob CN Schuldt. On the security of RC4 in TLS. In *USENIX Security Symposium*, pages 305–320, 2013.

[3] Bernhard Amann, Matthias Vallentin, Seth Hall, and Robin Sommer. Extracting certificates from live traffic: A near real-time SSL notary service. *Technical Report TR-12-014*, 2012.

[4] Blake Anderson, Subharthi Paul, and David McGrew. Deciphering malwares use of tls (without decryption). *Journal of Computer Virology and Hacking Techniques*, pages 1–17, 2016.

[5] arma. Obfsproxy: the next step in the censorship arms race. https://blog.torproject.org/obfsproxy-next-step-censorship-arms-race, 2012.

[6] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS with SSLv2. In *25th USENIX Security Symposium*, August 2016.

[7] Mike Belshe, Martin Thomson, and Roberto Peon. Hypertext transfer protocol version 2 (http/2). 2015.

[8] David Benjamin. Applying grease to tls extensibility. https://tools.ietf.org/html/draft-davidben-tls-grease-01, September 2016.

[9] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 535–552. IEEE, 2015.

[10] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 456–467. ACM, 2016.

[11] Lee Brotherston. Stealthier attacks and smarter defending with tls fingerprinting, 2015.

[12] Emil Burzo. No ability to use a supported (but not enabled) cipher suite. https://github.com/square/okhttp/issues/2698, 2016.

[13] Roger Dingledine and Jacob Appelbaum. How governments have tried to block tor. 2012.

[14] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[15] Chromium Developer Documentation. SPDY: An experimental protocol for a faster web. https://www.chromium.org/spdy/spdy-whitepaper, 2009.

[16] Tor documentation. Tor: Pluggable Transports. https://www.torproject.org/docs/pluggable-transports.html.

[17] Tor documentation. meek: pluggable transport, an obfuscation layer for tor designed to evade internet censorship. https://trac.torproject.org/projects/tor/wiki/doc/meek, 2018.

[18] Tor documentation. Snowflake: pluggable transport that proxies traffic through temporary proxies using webrtc. https://trac.torproject.org/projects/tor/wiki/doc/Snowflake, 2018.

[19] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. The security impact of https interception. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2017.

[20] Frank Ehlis. How to disable ssl ciphers in google chrome, 2013.

[21] David Fifield. Cyberoam firewall blocks meek by TLS signature. https://groups.google.com/forum/\#!topic/traffic-obf/BpFSCVgi5rs/, 2016.

[22] David Fifield. Fortiguard firewall blocks meek by TLS signature. https://groups.google.com/forum/\#!topic/traffic-obf/fwAN-WWz2Bk, 2016.

[23] David Fifield. Anticipating a world of encrypted SNI: risks, opportunities, how to win big . https://groups.google.com/d/msg/traffic-obf/UyaLc9jPNmY/ovNImK5HEQAJ, August 2018.

[24] David Fifield, Nate Hardison, Jonathan Ellithorpe, Emily Stark, Dan Boneh, Roger Dingledine, and Phil Porras. Evading censorship with browser-based proxies. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 239–258. Springer, 2012.

[25] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies*, 2015(2):46–64, 2015.

[26] John Geddes, Max Schuchard, and Nicholas Hopper. Cover your acks: Pitfalls of covert channel censorship circumvention. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 361–372. ACM, 2013.

[27] John Gilmore. Cracking DES: Secrets of encryption research. *Wiretap Politics & Chip Design*, 272, 1998.

[28] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kâafar. TLS in the wild: an internet-wide analysis of tls-based protocols for electronic communication. *CoRR*, abs/1511.00341, 2015.

[29] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. The SSL landscape: a thorough analysis of the X.509 PKI using active and passive measurements. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 427–444. ACM, 2011.

[30] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 65–79. IEEE, 2013.

[31] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged ssl certificates in the wild. In *Security and privacy (sp), 2014 ieee symposium on*, pages 83–97. IEEE, 2014.

[32] Psiphon Inc. Psiphon — uncensored internet access for windows and mobile. https://www.psiphon3.com, 2018.

[33] Sheharbano Khattak, Tariq Elahi, Laurent Simon, Colleen M Swanson, Steven J Murdoch, and Ian Goldberg. Sok: Making sense of censorship resistance systems. *Proceedings on Privacy Enhancing Technologies*, 2016(4):37–61, 2016.

[34] SSL Labs. HTTP client fingerprinting using SSL handshake analysis. https://www.ssllabs.com/projects/client-fingerprinting/, 2009.

[35] The Go Programming Language. cgo, 2018.

[36] Homin K Lee, Tal Malkin, and Erich Nahum. Cryptographic strength of SSL/TLS servers: current and recent practices. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 83–92. ACM, 2007.

[37] Fangfan Li, Abbas Razaghpanah, Arash Molavi Kakhki, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. lib erate,(n): A library for exposing (traffic-classification) rules and avoiding them efficiently. In *Proceedings of the 2017 Internet Measurement Conference*, pages 128–141. ACM, 2017.

[38] Colm MacCarthaigh. Enhanced domain protections for amazon cloudfront requests. https://aws.amazon.com/blogs/security/enhanced-domain-protections-for-amazon-cloudfront-requests/, 2018.

[39] Moxie Marlinspike. Amazon threatens to suspend signal's aws account over censorship circumvention. https://signal.org/blog/looking-back-on-the-front/, 2018.

[40] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: exploiting the ssl 3.0 fallback. *Security Advisory*, 2014.

[41] moxie0. Doodles, stickers, and censorship circumvention for Signal Android. https://signal.org/blog/doodles-stickers-censorship/, 2016.

[42] phobos. Update on internet censorship in Iran. https://blog.torproject.org/update-internet-censorship-iran, 2011.

[43] Lantern Project. Lampshade: a transport between Lantern clients and proxies. https://godoc.org/github.com/getlantern/lampshade.

[44] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. Studying tls usage in android apps. 2017.

[45] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[46] Eric Rescorla. Transport Layer Security (TLS) parameters. https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml, 2018.

[47] Ivan Ristić. HTTP client fingerprinting using SSL handshake analysis. https://blog.ivanristic.com/2009/06/http-client-fingerprinting-using-ssl-handshake-analysis.html, 2009.

[48] Ivan Ristić. sslhaf: Passive ssl client fingerprinting using handshake analysis. https://github.com/ssllabs/sslhaf, 2009.

[49] Runa. Ethiopia introduces deep packet inspection. https://blog.torproject.org/ethiopia-introduces-deep-packet-inspection, 2012.

[50] runa. Kazakhstan uses DPI to block Tor. https://trac.torproject.org/projects/tor/ticket/6140, 2012.

[51] runa. Uae uses DPI to block Tor. https://trac.torproject.org/projects/tor/ticket/6246, 2012.

[52] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen K Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. MD5 considered harmful today, creating a rogue CA certificate. In *25th Annual Chaos Communication Congress*, number EPFL-CONF-164547, 2008.

[53] statcounter. Browser version market share worldwide, 2016.

[54] Let's Encrypt Stats. Percentage of Web Pages Loaded by Firefox Using HTTPS. https://letsencrypt.org/stats/#percent-pageloads, 2018.

[55] Marc Stevens. Fast collision attack on MD5. *IACR Cryptology ePrint Archive*, 2006:104, 2006.

[56] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In *Annual International Cryptology Conference*, pages 570–596. Springer, 2017.

[57] Open Whisper Systems. Signal - private messenger. https://signal.org/.

[58] Inc. ToorCon. crack.sh — the world's fastest des cracker. https://crack.sh/.

[59] Michael Carl Tschantz, Sadia Afroz, Vern Paxson, et al. Sok: Towards grounding censorship circumvention in empiricism. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 914–933. IEEE, 2016.

[60] twilde. Knock Knock Knockin' on Bridges' Doors. https://blog.torproject.org/knock-knock-knockin-bridges-doors, 2012.

[61] Mathy Vanhoef and Frank Piessens. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security Symposium*, pages 97–112, 2015.

[62] Liang Wang, Kevin P Dyer, Aditya Akella, Thomas Ristenpart, and Thomas Shrimpton. Seeing through network-protocol obfuscation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 57–69. ACM, 2015.

[63] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *Annual international cryptology conference*, pages 17–36. Springer, 2005.

[64] Philipp Winter, Tobias Pulls, and Juergen Fuss. Scramblesuit: A polymorphic network protocol to circumvent censorship. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 213–224. ACM, 2013.

[65] Eric Wustrow, Colleen Swanson, and J Alex Halderman. Tapdance: End-to-middle anticensorship without flow blocking. In *23rd USENIX Security Symposium*, August 2014.

APPENDIX A
MULTIPLE FINGERPRINTS USAGE

We show the ease of using uTLS as compared to using the standard `crypto/tls` library (which provides no control over TLS). In this configuration, uTLS will mimic popular browsers until an unblocked one is found.

```go
utlsRoller, err := tls.NewRoller()
if err != nil {
   return err
}

conn, err := utlsRoller.Dial("tcp",
   "10.1.2.3:443", "golang.org")
if err != nil {
   return err
}
conn.Write([]byte("Hello, Golang!"))
```

Listing 1. Dialing with utls.Roller

```go
tlsConf := tls.Config{
   ServerName: "golang.org",
}

conn, err = tls.Dial("tcp",
   "10.1.2.3:443", &tlsConf)
if err != nil {
   return err
}
conn.Write([]byte("Hello, Golang!"))
```

Listing 2. Dialing with standard crypto/tls and no mimicry