When TLS Meets Proxy on Mobile

Joyanta Debnath[1](✉), Sze Yiu Chau[2], and Omar Chowdhury[1]

[1] The University of Iowa, Iowa City, USA
{joyanta-debnath,omar-chowdhury}@uiowa.edu
[2] The Chinese University of Hong Kong, Sha Tin, Hong Kong
sychau@ie.cuhk.edu.hk

Abstract. Increasingly more mobile browsers are developed to use proxies for traffic compression and censorship circumvention. While these browsers can offer such desirable features, their security implications are, however, not well understood, especially when tangled with TLS in the mix. Apart from vendor-specific proprietary designs, there are mainly 2 models of using proxies with browsers: TLS interception and HTTP tunneling. To understand the current practices employed by proxy-based mobile browsers, we analyze 34 Android browser apps that are representative of the ecosystem, and examine how their deployments are affecting communication security. Though the impacts of TLS interception on security was studied before in other contexts, proxy-based mobile browsers were not considered previously. In addition, the tunneling model requires the browser itself to enforce certain desired security policies (*e.g.*, validating certificates and avoiding the use of weak cipher suites), and it is preferable to have such enforcement matching the security level of conventional desktop browsers. Our evaluation shows that many proxy-based mobile browsers downgrade the overall quality of TLS sessions, by for example allowing old versions of TLS (*e.g.*, SSLv3.0 and TLSv1.0) and accepting weak cryptographic algorithms (*e.g.*, 3DES and RC4) as well as unsatisfactory certificates (*e.g.*, revoked or signed by untrusted CAs), thus exposing their users to potential security and privacy threats. We have reported our findings to the vendors of vulnerable proxy-based browsers and are waiting for their response.

Keywords: TLS interception · HTTP tunneling · Proxy-based browsers

1 Introduction

Smartphones have proliferated in the last decade, and consequently there has been a strong growth in Internet traffic powered by mobile devices. The high portability and mobility, however, often comes at a cost, in terms of limitations on bandwidth and latency. Unlike conventional Internet access, mobile data is typically *metered*, and services offer limited capacity within a specific period. Because of this, increasingly there are more browsers offering a so-called "data-saving" feature, which leverage a proxy to help cache and compress objects that

© Springer Nature Switzerland AG 2020
M. Conti et al. (Eds.): ACNS 2020, LNCS 12147, pp. 387–407, 2020.
https://doi.org/10.1007/978-3-030-57878-7_19

need to be sent to the mobile browser, with the aim of reducing mobile data consumption and in some cases lowering the latency as well.

There are other reasons for using proxy-based browser apps on mobile platforms. For example, some use the proxy to conceal their own IP addresses for privacy protection against potentially malicious Web servers. Some would use these apps to bypass *geo-blocking*, a technology commonly used to adjust and restrict contents based on estimations of the users' geolocation. Others rely on these apps to circumvent various forms of censorship [15,22,24,35,38].

Despite their desirable features, the use of proxy-based browsers is not without its complications, especially when the users' security and privacy are taken into considerations. In this paper, we set out to investigate the implications on security when using these browsers, specifically the scenario where they are entangled with TLS. As we will explain later, based on our investigations, there are primarily two ways of deploying proxies with mobile browsers, along with some other proprietary technologies. The first way follows the TLS interception model, where the proxy acts as an active man-in-the-middle (MITM) and establishes 2 TLS connections (one with the browser, one with the actual Web server). We note that the impact of TLS interception on security has been examined before in the context of anti-virus and parental control software [11], as well as network middleboxes in enterprise environments [14,37]. However, proxy-based browsers that follow the TLS interception model constitute another class of TLS-intercepting appliances that was not considered by previous work. In this model, the proxy is in charge of enforcing security policies (*e.g.*, avoiding the use of weak ciphers, and validating certificates). It is also desirable to have the two TLS connections exhibit some degree of symmetry in terms of the strength of their corresponding security parameters. Adapting some of the metrics proposed by previous work, we design experiments to evaluate the quality of the TLS connections established by these browsers. Another common way of deploying proxies with mobile browsers is through HTTP tunneling. In this model, the end-to-end nature of a TLS connection is preserved, and the proxy merely helps to relay traffic. Thus the browser apps themselves need to carefully enforce security policies, and ideally they should be as robust as their desktop counterparts. Given that major desktop browsers have gone through years of scrutiny from the security research community, we distillate some of the best practices and design experiments to determine whether the tunneling browser apps are offering adequate protection to their users.

To our surprise, we found that many proxy-based browsers accept weak ciphers, weak TLS versions, and vulnerable certificates offered by a Web server. One notable example is the UC Mini - Best Tube Mate & Fast Video Downloader app, which has more than a hundred million downloads, accepts legacy TLS versions (*e.g.*, TLSv1.0, SSLv3.0, and SSLv2.0), and broken ciphers (*e.g.*, RC4, and 3DES). These findings are worrisome, particularly when one takes into consideration that many users rely on these apps to circumvent censorship. When these apps do not provide an adequate level of security, an oppressive censor can tamper with the TLS connections to attack the users. We believe this research is beneficial in helping the community understand the risks associated with the current practices embraced by vendors of proxy-based mobile browsers, and we have shared our findings with the browser vendors so that they can and improve the overall security of proxy-based browsers.

2 Background

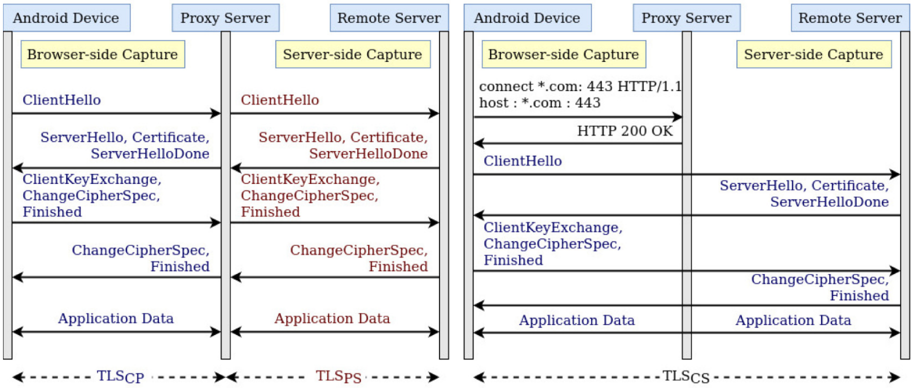
Here we explain technical details of the TLS interception and HTTP tunneling models, using TLSv1.2 handshake messages, employed by most of the proxy-based browsers studied in this paper. However, as we will explain later, some might use a proprietary protocol that is slightly different from these two models.

2.1 TLS Interception

TLS interception [19] is a common technique used to defeat the end-to-end security of TLS and allows the MITM to inspect contents transmitted between the client and the Web server. In this model, the proxy acts as an active MITM, so that some of the contents sent by the Web server could be cached and compressed before sending to the client. The cached objects might be reused later for other clients as well. In a typical setup of TLS interception, a trusted Certificate Authority (CA) certificate of the MITM is installed on the client's machine, so that any MITM-signed certificates will be accepted by the certificate validation procedure. As shown in Fig. 1a, the whole process starts with the client initiating a TLS handshake with a `ClientHello` sent to remote Web server. The proxy however captures and blocks this, and sends its own `ClientHello` to the Web server. The Web server replies back with `ServerHello`, `Certificates`, and `ServerHelloDone` handshake messages to the proxy. The proxy server should then validate the received certificate chain, and if the validation is successful, then it sends back to the client its own `ServerHello`, MITM-signed `Certificates`, and `ServerHelloDone` handshake messages. The MITM-signed certificate chain will then be validated by the client, and the rest of the handshake (*i.e.*, `ClientKeyExchange`, `ChangeCipherSpec`, `Finished`) happens for the two TLS sessions. Hence, through the proxy, two related TLS sessions (*i.e.*, TLS_{CP}, TLS_{PS}) are established between the client and the remote Web server, instead of one. The proxy server can now decrypt any incoming data (*i.e.*, `Application data`) from the client or the server, inspect, or even modify the data before encrypting again to forward it.

2.2 HTTP Tunneling

In HTTP tunneling, the client requests the proxy to relay a TCP connection to the Web server. In most cases, the tunnel is established using the `HTTP CONNECT` [RFC7231], though other HTTP methods can also be used depending on the setup. As shown in Fig. 1b, the client initiates by requesting a tunnel with `HTTP CONNECT`, and specify the port and the Web server that it wants to communicate


Android Device
Proxy Server
Remote Server
Browser-side Capture
Server-side Capture
ClientHello
ClientHello
ServerHello, Certificate, ServerHelloDone
ServerHello, Certificate, ServerHelloDone
ClientKeyExchange, ChangeCipherSpec, Finished
ClientKeyExchange, ChangeCipherSpec, Finished
ChangeCipherSpec, Finished
ChangeCipherSpec, Finished
Application Data
Application Data
TLSCP
TLSPS
Android Device
Proxy Server
Remote Server
Browser-side Capture
Server-side Capture
connect *.com: 443 HTTP/1.1
host : *.com : 443
HTTP 200 OK
ClientHello
ServerHello, Certificate, ServerHelloDone
ClientKeyExchange, ChangeCipherSpec, Finished
ChangeCipherSpec, Finished
Application Data
Application Data
TLSCS

(a) TLS Interception

(b) HTTP Tunneling

Fig. 1. Comparison of the two proxy deployment models. The yellow boxes illustrate our measurement setup. (Color figure online)

with. After receiving the request, the proxy server tries to establish a TCP connection with the Web server specified by the client. If the TCP connection is successful, the proxy server sends back to the client an `HTTP 200 OK`, indicating success. The client can then start communicating with the Web server, and the proxy server relays all the subsequent TCP stream between the client and the remote Web server, including the TLS handshake messages between the two sides, as well as the encrypted records. Because of this, the proxy cannot decrypt, read, or modify the contents of these messages, assuming proper cryptographic algorithms are being used. However, from the perspective of the Web server, it would appear that the proxy is the client, as the proxy's IP address would be used in the IP header. Compared to the TLS interception model, though there are also two TCP connections, only one TLS session (*i.e.*, $\boldsymbol{TLS_{CS}}$) is established.

3 Scopes and Methodologies

In this section, we define the scope of our study. We first discuss how we selected proxy-based mobile browsers apps, and then we describe the experiments used to evaluate their security.

3.1 Selection of Proxy-Based Mobile Browsers

We initially select a total of 36 proxy-based mobile browsers on *Android*; the full list is available at our website [12]. We focus on Android because it is currently the most popular operating system. As discussed before, proxy-based browsers are typically motivated by reduced mobile data consumption, privacy protection, and censorship circumvention. We thus looked for mobile browsers that are advertised with these keywords on app stores like Google Play and AppBrain. In addition to the popular ones, we also consider some relatively lesser known ones, in order to get a more comprehensive picture of the entire landscape.

3.2 Test Environment

Our experiment setup consists of three major components controlled by us: an Android device, a Linux laptop acting as a Wi-Fi access point (AP), and a TLS enabled test website. The goal of this test environment is to analyze all traffic between a proxy-based browser installed on the Android device, and the test website visited by the browser. In general, we refer the traffic between browser and proxy as *browser-side* and the traffic between proxy and test website as *server-side*. We configure the Android device to connect to the Wi-Fi AP provided by the Linux laptop, which allows us to capture browser-side traffic using common network analyzer tools (*e.g.*, Wirehark , and tshark). To capture server-side traffic, we run tshark on the Web server hosting the test website. The yellow boxes with blue borders in Fig. 1 illustrate this idea.

In some of the apps, the proxy-based mode (*e.g.* data-saving feature) is optional and not turned on by default. Hence for these apps, we switch on the relevant options prior to any of the experiments.

3.3 Identification and Classification of Proxy

Now we describe our approach for identifying the (IP address of) proxy server which is necessary to carry out our security evaluations. We also classify the browsers according to the model of their proxy deployment. We have written a Python program to automatically find out these information by analyzing the traffic captured from both browser-side and server-side.

Determining Proxy Server Address. A browser can automatically generate or receive extra traffic from different websites without user interaction. For instance, when a browser uses Google as its default search engine, additional traffic may be present due to auto-complete. Therefore, browser-side traffic tends to be noisy. To filter out the extra traffic and facilitate subsequent analysis, it is useful to know the IP address of the proxy server. Hence, we obtain this information by matching the browser-side and server-side traffic with some heuristics.

Since the server-side traces tend to be cleaner (highly unlikely to have additional visitors beyond our experiments), we can use the Unix epoch time to help match the traffic between the two sides. We use the following steps to automatically identify the proxy server address.

1. We find out the epoch times of the first (let this be $fTime$) and last (let this be $lTime$) frames of the server-side trace.
2. Since browser initiates the communication to server, and server sends data back to the browser, traffic of browser-side begins before the first frame of server-side, and traffic of browser-side ends after the last frame of server-side. We also find client-server communication tends to happen very quickly,

and empirically a 1-second threshold is enough to handle most cases. So we subtract 1 second from *fTime* and add 1 second with *lTime* to estimate the corresponding epoch time range for the client-side traffic.

3. Then, we separate destination addresses of the browser-side's trace and source addresses of the server-side's trace into two different sets.
4. If we find any common IP address in both of these sets, this IP address is the address of the proxy server. Otherwise, we know the IP addresses of the proxy server are different in these two traces. So, we go to step 5.
5. Since we have not found any exact match yet, we check whether any two IP addresses from these sets are from same network. In this step, we start matching with a 31-bit netmask and continue till a 28-bit netmask is used. When there is no match, we go to step 6.
6. In this step, we consider a 27-bit netmask down to a 20-bit netmask. In addition, we find out the locations of an IP address using the netaddr Python api. We match the ASN number, city name, region name, and country name from the locations of each pair of IP addresses. If this step still cannot find any match, we go to step 7.
7. Here, we consider netmask 19 to netmask 16 and location matching just like step 6. In addition to these two checkings, we consider the volume of traffic generated per IP address. If two IP addresses from our two sets have almost similar volume (for a particular TCP stream), we conclude both of these IP addresses belong to the same proxy server. Otherwise, we go to step 8.
8. If none of our above heuristics works, we manually analyze the two traces to determine what happened.

Identifying Two Models of Proxy. We classify these proxy-based browsers into two categories according to the role of their proxy servers : HT browser and TI browser, where 'HT' stands for **HTTP tunneling**, and 'TI' stands for **TLS interception**. Proxy server of an HT browser uses HTTP tunneling to create a TLS session between the browser and the website. As shown in Fig. 1b, these browsers most commonly use the `HTTP CONNECT` method to initiate the tunnel.

Moreover, our Python program looks at the certificate chain received from the identified proxy server in browser-side traffic. In the HT model, the proxy would forward the server's certificate chain to the browser without any modifications. Therefore, if the certificate chain received on the browser-side is exactly the same as the one that server sent, we classify the browser as an HT browser. If the certificate chain is different from the server's original certificate chain, then we classify the browser as a TI browser. From our list of browser applications, we found 18 to be HT browsers, and 16 to be TI-like browsers. We note that some TI-like browsers use a slightly unorthodox approach for TLS_{CP} (*i.e.*, TLS session between browser and proxy server). For example, we found 3 browsers are performing relaxed variants of TI: BROXY Browser and X Web Proxy establish TLS connections only for server-side traffic, and no TLS is used for browser-side; Unblock Sites forces HTTP on both sides and sends all data in plaintext, even if the user explicitly requests HTTPS (*i.e.*, by typing `https://` in the URI). Therefore, these browsers are listed as variants of TI browsers [12].

Finally, we categorize 1 browser as *Unidentified* since we cannot observe its proxy-based traffic. The 'Turbo Mode' of Yandex Browser is supposed to leverage proxy servers for data compression, however, we were unable to activate this feature even when we downgrade our uplink and downlink bandwidth to 10kbps. For this reason, we exclude Yandex Browser from our subsequent security evaluations and focus on the remaining 34 apps.

3.4 Security Evaluations

We run multiple experiments to evaluate the strength of overall TLS session between a proxy-based browser and our test website. To do so, we capture traffic from both browser-side and server-side, and perform automatic analysis with a Python program that performs the following security evaluations. The first three security evaluations focuses on TI browsers whereas the last three are applicable for both TI and HT browsers.

Evaluation 1 : Maintaining Strength of Certificate Parameters. The strength of a TLS session depends highly on important certificate parameters such as signature key length and signature hash algorithm. The use of short key length (*e.g.*, RSA-512) and deprecated hash algorithms (*e.g.*, MD5) can pose serious threats to a TLS session [RFC6151]. Therefore, it is recommended that the proxy server use strong parameters in its MITM-signed certificates for browser-side TLS session, or at least it should maintain the same strength as the server's certificates. This is primarily a requirement for TI but not HT browsers.

To evaluate this property, we consider different signature hash algorithms (*e.g.*, SHA256, SHA384, and SHA512), and different certificate key lengths (*e.g.*, RSA-2048, RSA-3072, and RSA-4096). Then, we consider all possible combinations of these certificate parameters to obtain 9 valid certificates from different trusted commercial CAs. For each of these 9 certificates, we load a certificate chain and its corresponding private key to the test Web server and visit the test website from each of the TI browsers. We find out the signature hash algorithms and signature key lengths used in the MITM-signed certificates and investigate whether the proxy server mirrors or downgrades the signature hash algorithms and certificate key lengths with respect to the test Web server's certificates.

In addition to signature hash algorithm and signature key length, we also analyze the validation level of the MITM-signed leaf certificate. We check whether the MITM-signed leaf certificate is `Extended Validated (EV)`, `Organizational Validated (OV)`, or `Domain Validated (DV)`. EV certificates are the most trus-ted ones, as CAs are supposed to issue this type of certificate only after thorough validation of Web server's identity. On the other hand, DV certificates are issued very quickly after some basic validation process, and it offers the least amount of trust among these three. It is preferable that the proxy server should not downgrade the trust level in the MITM-signed certificate compared to the server certificate. For instance, if the server-signed certificate is EV certificate, the MITM-signed certificate should also be EV. For this experiment, we visit Twitter's website (https://mobile.twitter.com/) to explore the validation level of the MITM-signed certificate with respect to Twitter's EV certificate. The validation level of a certificate can be found by inspecting the policy identifier value of the policy extension.

Evaluation 2 : Mirroring TLS Version and Strength of Cipher Suites. This is another desirable property for TI class browsers. The TLS version used by the browser-side TLS_{CP} should not be weaker than that of the server-side TLS_{PS}. This property is necessary to ensure that the proxy is not downgrading the protocol version. We enumerate different versions of TLS (and SSL) and see if the TLS versions negotiation observed in the `ClientHello` and `ServerHello` pairs from both sides match with each other and determine whether the proxy mirrors, upgrades, or downgrades the TLS version.

In addition to the TLS version, when the proxy receives a set of cipher suites proposed by the browser through `ClientHello`, the `ClientHello` sent by the proxy to the remote Web server should offer a comparable set of cipher suites, or it should at least ensure no weak or insecure cipher suites are being offered. Similarly, after the Web server chose a particular cipher suite to use for TLS_{PS}, the proxy should choose one with similar strength, if not exactly the same, for TLS_{CP}. This property is important to ensure similar strength of key exchange algorithms, ciphers, and message authentication code are used in both sides of the proxy (*i.e.*, TLS_{CP} and TLS_{PS}). We find out the sets of cipher suites being negotiated by monitoring the pairs of `ClientHello` and `ServerHello` in TLS_{CP} and TLS_{PS}. Then, we can investigate whether the proxy offers better/weaker cipher suites to the Web server.

Evaluation 3 : Validation of Proxy Certificates. For TI browsers, they need to properly validate certificates coming from their proxies, unless they employ other means for authenticating the proxy servers. Without a robust certificate chain validation, it might be possible for an MITM to perform impersonation attacks against TLS_{CP} and intercept the data exchanged between the browser and its proxy. For this evaluation, we deploy the open source mitmproxy on our Linux wireless AP, to inject invalid certificates to TLS_{CP} when it gets established and see if any of the TI browsers would accept such certificates. Considering the certificate chains of the proxies, we are also interested in seeing the length of the chains and for how long would they be valid for.

Evaluation 4 : Avoiding Weak Cipher Suites. If a Web server is configured to only use weak cipher suites (*e.g.*, those using flawed ciphers like RC4, 3DES and other export-grade ciphers), the proxy (TI browsers) or browser (HT browsers) should not establish a TLS session with it, or some warning messages should be shown to the users. This property is required to provide some basic guarantees on the strength of the overall TLS session between the browser and the remote website. The users might be misled into a false sense of security if the apps silently establish TLS sessions with weak ciphers being used.

In HT browsers, weak cipher suites should be avoided by the browser application itself since the proxy does not actively interfere with the communication with the Web server. However, for TI browsers, the proxy needs to enforce policies regarding weak ciphers. In this experiment, we initially configure the test website with different cipher suite parameters (*e.g.*, OpenSSL's `HIGH`, `MEDIUM`, `LOW`, or `EXPORT`). `LOW` and `EXPORT` level cipher suites are expected to be blocked by the proxy server or the browser because these cipher suites should not be used due to their weaknesses. We also try some additional weak cipher suites involving vulnerable ciphers (*i.e.*, RC4 and 3DES) and modes that are known to be tricky to implement (*i.e.*, block cipher in CBC mode with HMAC-SHA1 and HMAC-MD5) in the test Web server to see if any of the HT and TI browsers consider these acceptable.

Evaluation 5 : Validation of Server's Certificate Chain. In TLS handshake, server sends a certificate chain with `ServerHello` message as a reply to Client's `ClientHello` message. The leaf certificate of this certificate chain contains server's public key which is signed by some trusted CA. The client after receiving this certificate chain verifies it to make sure that the client is communicating with the correct server, not an impersonator. The validation process should check the trustworthiness of issuers on the chain, the cryptographic signatures, hostname, revocation status, and validity period. Otherwise, an attacker can exploit some of the missing checks for potential attacks.

The browser (in the HT model) or its proxy (in the TI model) has to perform the aforementioned checks on the received certificate chain, and if the validation fails, it should decline to establish a TLS session, or show appropriate warning messages to the user. To check whether the apps adequately perform such validations, we use the following types of certificates in our experiments: **self-signed** certificates, certificates signed by an untrusted issuer (**Custom CA**), certificates with **invalid signatures** and **incorrect common names**, **revoked** certificates, and **expired** certificates. We configure our test Web server to use these certificates, one at a time, and visit the test website from each of the browsers to determine the robustness of their certificate chain validation process.

Unlike previous work [11], for TI browsers, we cannot install a custom trusted CA to the proxy servers to enable fine-grained experiments. Instead, we have to rely on common trusted commercial CAs to issue our test certificates, and hence the combination of certificate issues that we can experiment with, especially regarding X.509v3 extensions, are restricted by the certificate issuance policies of the commercial CAs. While certificates with incorrect common name was easily obtainable from commercial CAs, we manually modify the signature of a valid certificate to obtain test certificates with invalid signatures. For revoked certificates, we have waited for 2 months to allow the revocation information to get properly disseminated before testing. Considering HT browsers, we also attempt to install a trusted CA certificate on the Android test device, and if the browsers trust the system CA store, additional fine-grained experiments regarding problematic certificate can be performed.

Evaluation 6 : Avoiding Weak TLS Versions. If a remote website only accepts legacy TLS (*e.g.*, versions older than TLSv1.1), the browser or its proxy server should prevent the TLS session from being established, or show appropriate warning message to the user. This is useful in providing the user some guarantees on the overall strength of the TLS sessions. Historic versions of TLS often lack support for modern ciphersuites, and might be susceptible to different kinds of MITM and downgrade attacks [RFC7568, RFC6176].

For the same reason explained in Evaluation 4, this should be performed by the browser application (in case of HT browsers), or by the proxy server (in case of TI browsers). We perform this experiment by configuring the aforementioned weak TLS versions one-by-one in our test Web server. Then, for each of these TLS versions, we visit the test website from each of the browsers and see if the TLS sessions can be successfully established despite the weak TLS versions. This is more reliable than simply monitoring the `ClientHello` messages for protocol version numbers, as the browser/proxies might have fallback/retry mechanisms upon encountering a server with incompatible versions.

4 Findings

4.1 Commercial CA Certificate Issuance Policy

The variety of experiments regarding problematic certificates that we can perform depends on the issuance policy of the trusted issuers. Hence, we set out to explore the possibility of obtaining certificates with weak algorithms and/or bad parameters through commercial CAs. We examined the certificate issuance policies of 11 well-known CAs listed in online surveys[1] and encyclopedia (*e.g.*, Wikipedia). We found that none of the CAs agreed to issue certificates with weak signature hash algorithms (*e.g.*, SHA1, MD5) or short RSA keys (*e.g.*, 512 bits, 1024 bits). At minimum, the issuance policies of these CAs require certificates to use SHA256 with a RSA modulus of at least 2048-bit long, which is coherent with the baseline requirements published by the CA/Browser Forum [1]. While it is fortunate that the commercial CAs we considered all have a somewhat high bar in terms of what kind of certificates they are willing to issue, this also means that for Evaluation 5 (Validation of Server's Certificate Chain), we are unable to perform fine-grained analysis of how the certificate validation performed by the TI proxies behave under different problematic certificates.

4.2 Maintaining Strength of Certificate Parameters

We found that the certificates issued by the proxies of the 11 TI browsers all use a fixed signature hash algorithm and RSA key length, even when the Web server itself uses certificates of longer hashes/keys. See columns 2 & 3 of Table 1 for the results. Our experiments include different certificates with varying signature hash algorithms (*i.e.*, SHA-256/384/512) and different key lengths (RSA with 2048/3072/4096-bit modulus). However, all of these TI browsers use SHA256 as the signature hash algorithm, and nine of these use 2048-bit RSA modulus in their MITM certificates. The remaining one, Upx Browser uses a 1024-bit RSA

[1] https://w3techs.com/technologies/overview/ssl_certificate/all.

Table 1. Results of security evaluations 1–3.

Columns 2–4 show the fixed parameters of proxy certificates, irrespective of the server certificates.
Column 5 shows the TLS version of TLS_{CP}, irrespective of TLS_{PS}.
Columns 6–7 show results of Security Evaluation 2.
HA = Hash Algorithm SPK = Size of Public Key TL = Trust Level Ver = Version
CSD = Cipher Suites Different CV = Certificate Validity MCV = MITM Certificate Validation

Browser Name	HA in TLS_{CP}.cert	SPK in TLS_{CP}.cert	TL of TLS_{CP}.cert	Ver of TLS_{CP}	CSD	Proxy Selected Cipher Suite ‡	Issuer CV †	Leaf CV	MCV
Aloha Browser	SHA256	RSA-2048	OV /DV	TLSv1.2	Yes	0xc030	—	10 yrs	B
Aloha Browser Lite	SHA256	RSA-2048	OV /DV	TLSv1.2	Yes	0xc030	—	10 yrs	B
Google Chrome	SHA256	RSA-2048	OV	GQUIC46	Yes	#	4.5 yrs	3 mths	B
Opera	SHA256	RSA-2048	OV /DV	TLSv1.2	Yes	0xc02f	—	1 mth	W
Opera Beta	SHA256	RSA-2048	OV /DV	TLSv1.2	Yes	0xc02f	—	1 mth	W
Private Browser	SHA256	RSA-2048	OV /DV	TLSv1.2	Yes	0xc030	5 yrs	3 mths	B
Puffin Browser	SHA256	RSA-2048	DV	TLSv1.2	Yes	0xc030	20 yrs	2 mths	B
Tenta Browser	SHA256	RSA-2048	DV	TLSv1.2	Yes	0xc030	—	1 mth	B
Tunnel Browser	SHA256	RSA-2048	DV	TLSv1.2	Yes	0xc02f	5 yrs	1 mth	W
Unblock Website..	SHA256	RSA-2048	DV	TLSv1.2	Yes	0xc02f	5 yrs	1 mth	B
UPX Browser	SHA256	RSA-1024	OV /DV	TLSv1.3	Yes	0x1301	10 yrs	2 mths	B

‡ 0xc030 = TLS-ECDHE-RSA-WITH-AES256-GCM-SHA384;
0xc02f = TLS-ECDHE-RSA-WITH-AES128-GCM-SHA256;
0x1301 = TLS-AES128-GCM-SHA256
= Curve25519 for key exchange, and AES-GCM for authenticated encryption
† '—' means no issuer certificates were sent.

modulus. All in all, **none of the these apps mirrored the strength of the certificate parameters offered by the Web server**.

We also evaluated whether the MITM certificates maintain the trust level of server certificates. In our findings (column 4 of Table 1), the proxies of Puffin Browser, Unblock Website Browser, Tenta Browser, and Tunnel Browser send DV certificates to the browser even though the remote website (https://mobile.twitter.com) sends an EV certificate to the proxy. Google Chrome's proxy use an OV certificate. For rest of the TI browsers, MITM certificates do not have any policy extension fields. Hence, we can only confirm that these certificates are not EV, but cannot determine whether they are OV or DV. According to our findings, **none of the proxy servers use the most trusted EV certificates for TLS_{CP}**; additionally, users are not warned about such discrepancies.

4.3 Mirroring TLS Version and Strength of Cipher Suites

We found that the TLS version of TLS_{CP} for 10 TI browsers is fixed at TLSv1.2, regardless of the version used by TLS_{PS}. Google Chrome uses GQUIC to replace TLS_{CP}. To our pleasant surprise, UPX Browser, uses TLSv1.3 for its TLS_{CP}. Similarly, the cipher suite selected for TLS_{CP} also appears to be fixed, irrespective of what is chosen for TLS_{PS}, though different browsers have different preferences over possible cipher suites. Fortunately, the cipher suites chosen are all reasonably strong and with the property of forward secrecy. These results can be found in columns 5–7 of Table 1.

4.4 Validation of Proxy Certificates

Regarding certificate validation during the establishment of TLS_{CP}, we found that none of the 11 TI browsers silently accept the MITM-signed certificates injected by mitmproxy; 8 of them outright reject the certificates, and the remaining three (Opera, Opera Beta and Tunnel Browser) display warning messages prompting the user to decide whether to continue or not. See column 10 of Table 1 for the results. Moreover, columns 8–9 of Table 1 present the findings on the certificate chain used by the TI proxies. Five of the TI proxies send only a single certificate for authentication purposes, and the others send a chain of two certificates. Most of the proxy certificates (leaf of the chain) have a short validity period of a few months, and the only exceptions are Aloha Browser and Aloha Browser Lite, where the proxy certificates are valid for 10 years (column 9 of Table 1). For the proxies that sent a chain of two certificates, all of the issuer certificates are valid for more than a few years, with Puffin Browser having the longest validity period of 20 years (column 8 of Table 1).

4.5 Avoiding Weak Cipher Suites

For this evaluation, we configure our test Web server with different cipher suites as described in Sect. 3.4. When we use OpenSSL's HIGH and MEDIUM level cipher suites (consisting of key lengths greater than or equal to 128 bits), all TI and HT browsers successfully established TLS sessions. On the other hand, none of the browsers were willing to established TLS sessions with the test Web server when it is using OpenSSL's LOW and EXPORT level cipher suites (consisting of key length less or equal to 64 bits). See columns 2–5 of Table 2 for the results.

Interestingly, a more fine-grained experiment with cipher suites revealed some subtle issues that are worth considering. When we configure the test Web server with cipher suites involving the use of algorithms like **SHA1** and **3DES** that are considered to be weak, **all of the 34 browsers tested turn out to be willing to establish TLS sessions with the remote website**, without showing any warnings to their users warning. Moreover, we found that Tunnel Browser, and the proxy servers of UC Mini and Unblock Website Browser, are willing to **accept without warning cipher suites involving weak algorithms like RC4 and MD5**. These behaviors are consistent with the `ClientHello` requests observed on the Web server side. See columns 6–9 of Table 2 for the results. RC4 has irreparable weaknesses that can open door to a variety of attacks [3,16,36], 3DES is susceptible to birthday attacks due to its small block size [7], and cipher suites using MD5 and SHA1 are either using the flawed RC4 stream cipher, or block ciphers in CBC mode which has proven to be tricky to implement and are continuously haunted by padding oracle attacks [2,18,28].

4.6 Validation of Server's Certificate Chain

Certificate chain validation ensures that a TLS session was established with the intended entity, given that the claimed identity was verified and vouched by some trusted authorities. The results of this evaluation can be found in Table 2.

Table 2. Results of security evaluation 4–5.

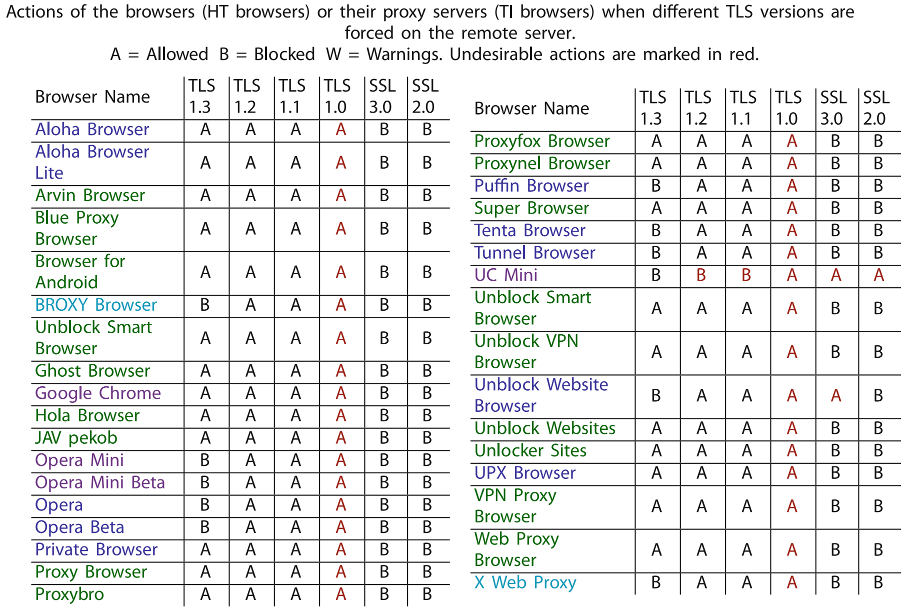
Left half of the table shows actions taken by the browsers (HT browsers) or proxies (TI browsers) on different strengths of cipher suites configured in the Web server.
Right half of the table shows actions taken by the browsers (HT browsers) or proxies (TI browsers) when various invalid certificates are sent from the remote server.
A = Allowed B = Blocked W = Warnings H = HIGH M = MEDIUM L = LOW ET = EXPORT
SS= Self-Signed SM = Signature-Mismatch WCN = Wrong Common Name RV = Revoked
EX = Expired CCA = Custom CA Undesirable actions are marked in red.

Browser Name	H	M	L	ET	RC4	MD5	3DES	SHA1	SS	SM	WCN	RV	EX	CCA
Aloha Browser	A	A	B	B	B	B	A	A	W	W	W	A	W	W
Aloha Browser Lite	A	A	B	B	B	B	A	A	W	W	W	A	W	W
Arvin Browser	A	A	B	B	B	B	A	A	B	B	B	A	B	B
Blue Proxy Browser	A	A	B	B	B	B	A	A	B	B	B	A	B	B
Browser for Android	A	A	B	B	B	B	A	A	W	W	W	A	W	W
BROXY Browser	A	A	B	B	B	B	A	A	A	A	A	A	A	A
Unblock Smart..	A	A	B	B	B	B	A	A	B	B	B	A	B	B
Ghost Browser	A	A	B	B	B	B	A	A	W	W	W	A	W	W
Google Chrome	A	A	B	B	B	B	A	A	W∗	W∗	W∗	A	W∗	W∗
Hola Browser	A	A	B	B	B	B	A	A	W	W	A	A	B	W
JAV pekob	A	A	B	B	B	B	A	A	W	W	W	A	W	W
Opera Mini	A	A	B	B	B	B	A	A	W	B	W	B	B	W
Opera Mini Beta	A	A	B	B	B	B	A	A	W	B	W	B	B	W
Opera	A	A	B	B	B	B	A	A	W	W	W	A	W	W
Opera Beta	A	A	B	B	B	B	A	A	W	W	W	A	W	W
Private Browser	A	A	B	B	B	B	A	A	B	B	W	A	W	B
Proxy Browser	A	A	B	B	B	B	A	A	B	B	B	A	B	B
Proxybro	A	A	B	B	B	B	A	A	W	W	W	A	W	W
Proxyfox Browser	A	A	B	B	B	B	A	A	W	W	W	A	W	W
Proxynel Browser	A	A	B	B	B	B	A	A	W	W	W	A	W	W
Puffin Browser	A	A	B	B	B	B	A	A	W	W	W	A	W	W
Super Browser	A	A	B	B	B	B	A	A	W	W	A	A	A	W
Tenta Browser	A	A	B	B	B	B	A	A	B	B	B	A	B	B
Tunnel Browser	A	A	B	B	A	A	A	A	A	A	A	A	A	A
UC Mini	A	A	B	B	A	A	A	A	B	B	B	A	B	B
Unblock Site Browser	A	A	B	B	B	B	A	A	W	W	A	A	A	W
Unblock VPN Browser	A	A	B	B	B	B	A	A	W	W	A	A	A	W
Unblock Website..	A	A	B	B	A	A	A	A	A	A	A	A	A	A
Unblock Websites	A	A	B	B	B	B	A	A	B	B	B	A	B	B
Unlocker Sites	A	A	B	B	B	B	A	A	B	B	B	A	B	B
UPX Browser	A	A	B	B	B	B	A	A	B	B	B	A	B	B
VPN Proxy Browser	A	A	B	B	B	B	A	A	B	B	B	A	B	B
Web Proxy Browser	A	A	B	B	B	B	A	A	W	W	W	A	W	W
X Web Proxy	A	A	B	B	B	B	A	A	A	A	A	A	A	A

∗ In this case, proxy gets inactive, and TLS session is continued directly between browser and remote server.

When the server certificate is *self-signed*, signed by an *untrusted issuers (Custom CA)*, or has an *invalid signature*, we found that all the HT browsers would either reject it and terminate the TLS connection, or show warning messages to the user before continuing. However, we noticed that four HT browsers (Hola Browser, Super Browser, Unblock Site Browser, and Unblock VPN Browser) would **accept server certificates with an *incorrect common name*** without showing any warnings to the users. The rest of the HT browsers all reject certificates with incorrect common name. Additionally, Super Browser, Unblock Site Browser, and Unblock VPN Browser also **appear to accept *expired certificates***.

Table 3. Results of security evaluation 6.



Actions of the browsers (HT browsers) or their proxy servers (TI browsers) when different TLS versions are forced on the remote server.
A = Allowed B = Blocked W = Warnings. Undesirable actions are marked in red.

Browser Name	TLS 1.3	TLS 1.2	TLS 1.1	TLS 1.0	SSL 3.0	SSL 2.0
Aloha Browser	A	A	A	A	B	B
Aloha Browser Lite	A	A	A	A	B	B
Arvin Browser	A	A	A	A	B	B
Blue Proxy Browser	A	A	A	A	B	B
Browser for Android	A	A	A	A	B	B
BROXY Browser	B	A	A	A	B	B
Unblock Smart Browser	A	A	A	A	B	B
Ghost Browser	A	A	A	A	B	B
Google Chrome	A	A	A	A	B	B
Hola Browser	A	A	A	A	B	B
JAV pekob	A	A	A	A	B	B
Opera Mini	B	A	A	A	B	B
Opera Mini Beta	B	A	A	A	B	B
Opera	B	A	A	A	B	B
Opera Beta	B	A	A	A	B	B
Private Browser	A	A	A	A	B	B
Proxy Browser	A	A	A	A	B	B
Proxybro	A	A	A	A	B	B
Proxyfox Browser	A	A	A	A	B	B
Proxynel Browser	A	A	A	A	B	B
Puffin Browser	B	A	A	A	B	B
Super Browser	A	A	A	A	B	B
Tenta Browser	B	A	A	A	B	B
Tunnel Browser	B	A	A	A	B	B
UC Mini	B	B	B	A	A	A
Unblock Smart Browser	A	A	A	A	B	B
Unblock VPN Browser	A	A	A	A	B	B
Unblock Website Browser	B	A	A	A	A	B
Unblock Websites	A	A	A	A	B	B
Unlocker Sites	A	A	A	A	B	B
UPX Browser	A	A	A	A	B	B
VPN Proxy Browser	A	A	A	A	B	B
Web Proxy Browser	A	A	A	A	B	B
X Web Proxy	B	A	A	A	B	B

For all the TI browsers except for Unblock Website Browser and Tunnel Browser, these 5 types of invalid certificates are either outright rejected by their corresponding proxies, or a warning is displayed to their users.

For revoked certificates, we found that out of all the browsers tested, only Opera Mini and Opera Mini Beta would take the revocation status of certificates into consideration, and reject revoked certificates. This is particularly interesting because Opera and Opera Beta, which are from the same vendor and are supposed to be more full-featured, **do not seem to reject revoked certificates**.

Finally, for the two TI-O browsers (BROXY Browser and X Web Proxy), which uses TLS only for server-side traffic but not for browser-side, their certificate chain validation also appears to be very weak, as the proxy servers do not reject any of the 6 types of invalid certificates.

4.7 Avoiding Weak TLS Versions

For this evaluation, when the test server is configured to use TLSv1.2 or TLSv1.1, only UC Mini declined to communicate, while all the other browsers established TLS sessions without errors. Additionally, we noticed that quite **a few proxy-based browsers are still reluctant to support TLSv1.3**. See column 2–4 of Table 3 for the detailed results.

We have also found that **all the browsers continue to support the twenty year old TLSv1.0**. On the other hand, **SSLv3.0 is blocked by all the browsers except UC Mini and Unblock Website Browser**. This is interesting since UC Mini's proxy server does not support TLSv1.2 and TLSv1.1, but instead supports much older and weaker SSLv3.0 and TLSv1.0. We suspect that this is due to a lack of software upgrade and maintenance for these proxies.

Since SSLv2.0 is not supported by any reasonably recent versions of Apache Web server, we have not performed similar experiments with SSLv2.0. Instead, we used the Qualys SSL Client Test[2] to check the list of TLS versions that are supported by the proxy-based browsers. For HT browsers, such a test would reflect the configuration of the browser itself, and for TI browsers, this would effectively be testing the proxies. Through this additional experiment, we found that the **proxy server of UC Mini also supports SSLv2.0**, the usage of which is deprecated since 2011 [RFC6176].

5 Discussions

In this section, we discuss the implications of our findings, as well as some of the limitations of our experiments.

5.1 Browsers with No/Broken TLS

First of all, some proxy-based browsers are **effectively not benefiting from TLS at all**. For example, Unblock Sites strips TLS, and for the two TI-O browsers (Broxy Browser and X Web Proxy), they do not use TLS between the browser and the proxy, and certificate validation for TLS_{PS} is so weak that an impersonation+MITM attack can be mounted against it, which basically renders the TLS useless. Some other problematic browsers in this category include the Unblock Website Browser and Tunnel Browser, both of which have a fairly good TLS_{CP} that uses TLSv1.2 and a reasonably strong ephemeral cipher suite providing forward secrecy, but the TLS_{PS} certificate validation is abysmal and susceptible to impersonation+MITM attacks. Notice that these apps have hundreds to several hundred thousands of downloads. If users of these apps rely on them to exchange confidential data, there could be serious repercussions.

5.2 Leniency in Certificate Validation

Additionally, our experiments have revealed some other subtle unwarranted leniencies in how the browser apps (and their proxies) validate server certificates. Some of them **do not check for common names, opening doors to potential impersonation attacks**. One of the offenders, Hola Browser, had more than 50 million downloads, leaving a large number of users potentially vulnerable.

Moreover, some of the browsers, including ones that have garnered more than hundred thousand downloads (*e.g.*, Super Browser), do not reject expired certificates. While this does not seem immediately alarming, skipping the expiration

[2] https://www.ssllabs.com/ssltest/viewMyClient.html.

check is less than ideal for following reasons. First, accepting expired certificates means old certificates whose private keys might have been leaked and exposed (*e.g.*, through OpenSSL heartbleed bug) can still be used by an attacker. Second, given the support for revocation checks remains questionable, a phenomenon also observed by our experiments, there is a new trend of favoring short-lived certificates [25,34] instead of conventional revocation mechanisms (*e.g.*, CRLs and OCSP) that are considered to be heavyweight and not scaling well. In such a case, the ability to prevent certificates with leaked private keys from functioning again critically hinges on the browsers rejecting expired certificates, and hence we recommend browser vendors to **properly implement expiration checks**.

5.3 Weak Cipher Suites, TLS Versions and RSA Parameters

We have also noticed that **usage of weak cipher suites is not universally banned** in the 34 browsers we have studied. For example, 3 of them still support usage of RC4. RC4 is a stream cipher found to exhibit undesirable statistical biases in its key stream, which leads to a variety of attacks [3,16,36]. Major desktop browser vendors have disabled usage of RC4 for some years, and usage of RC4 have since been deprecated [RFC7465]. Moreover, 3DES is also considered weak, especially after the emergence of SWEET32 [7], a birthday attack exploiting its relatively short (64-bit) block size. NIST has since updated its guidelines to restrict the use of 3DES to encrypting not more than 2^{20} blocks (8 MB) of data under one key bundle (made of 3 unique 56-bit keys) [4], which is well within reach of a Web session involving a large amount of multimedia contents. Recently, NIST has announced usage of 3DES is deprecated and will be disallowed after year 2023 [5]. Interestingly, at the time of writing, we have seen that all the tested proxy-based browsers still support 3DES. **We hence recommend browser vendors to consider disabling support for RC4 as soon as possible, and follow the NIST guidelines on phasing out support for 3DES in the near future**.

On the other hand, there exists a series of research on reducing the complexity and monetary costs for finding SHA1 and MD5 collisions to within reach of resourceful adversaries [20,29–32,39,40], and vendors of major desktop browsers have already been rejecting SHA1 and MD5 certificates. However, their use as HMAC in TLS is not immediately problematic [RFC6151], as the security argument for HMAC does not depend on the collision resistance of the hash function [6]. The problem of cipher suites involving HMAC-SHA1 and HMAC-MD5 is that all of them involve either the irreparably flawed RC4 cipher, or block ciphers under the CBC mode, which when paired with the MAC-then-encrypt design choice embraced by TLS, has proven to be tricky to implement and leads to a variety of attacks [2,18,28]. TLSv1.2 has since introduced new cipher suites with authenticated encryption (*e.g.*, AES under GCM) [RFC5288], and TLSv1.3 has dropped all CBC-mode ciphers [RFC8446]. It is **advisable to consider removing support for such cipher suites** in the future, or **at least display warnings to the users** regarding these problematic cipher suites.

Apart from the issues related to problematic ciphers, historic versions of TLS like TLSv1.0 and TLSv1.1 are also found to have design flaws, for example, the SLOTH attack [8] demonstrates how to exploit transcript collision, resulted from the hash collision due to SHA1 and MD5, for breaking authentication in TLSv1.0 and TLSv1.1. On top of that, TLSv1.0 deployments can also be vulnerable to the BEAST attack [13], especially if the $1/1-n$ split client-side mitigation is not being implemented. As the result of which, vendors of major desktop browsers have all agreed to phase out support for TLSv1.0 and TLSv1.1 in 2020 [9,33], and some industry standardization body has already deprecated the use of TLSv1.0 [26]. There is also an IETF draft proposing to deprecate the use of TLSv1.0 and TLSv1.1 [23]. Consequently, we recommend vendors of proxy-based browsers to **consider following suite in phasing out support for TLSv1.0 soon, and TLSv1.1 as well in the near future**. For SSLv3.0 and SSLv2.0, both of them have already been deprecated due to numerous issues [RFC7568, RFC6176] and we recommend UC Mini and Unblock Website Browser to **stop supporting SSLv3.0 and SSLv2.0**.

In the context of TI class proxy-based browsers, this class of offense is particularly worrisome, as the use of historic versions of TLS and weak cipher suites for TLS_{PS} is transparent to the users, especially when TLS_{CP} itself is using reasonably good algorithms, which can potentially lead to a false sense of security. To the very least, **there should be warning messages delivered to the users when the quality of TLS_{PS} is subpar**.

Additionally, while UPX Browser is using TLSv1.3 with a reasonably strong cipher suit for TLS_{CP}, its proxy certificate has only a 1024-bit long RSA modulus. NIST has already recommended against usage of RSA modulus shorter than 2048 bits [5], and browsers like Firefox have already been phasing out support for certificates with 1024-bit RSA modulus [41]. Hence, **we recommend UPX Browser to upgrade its proxy certificate to use a longer RSA modulus**.

5.4 Asymmetry of TI Browsers

Finally, we note that for TI browsers, strength of the certificates (in terms of size of RSA modulus and hash algorithms) and cipher suites used by their TLS_{CP} and TLS_{PS} are often not mirrored. This can lead to two contrasting problems. First, as discussed in Sect. 5.3, a good quality TLS_{CP} without any warning messages could potentially mask the problem of a low quality TLS_{PS} (*e.g.*, bad certificates or broken ciphers), leading to a **false sense of security**. On the other hand, if a Web server is configured to use strong cipher suites and certificates with long RSA modulus, the fixed parameters for TLS_{CP} as presented in Table 1 can be seen as downgrading the overall quality of the TLS sessions. **A potentially better approach is to choose matching parameters for TLS_{CP} based on the outcome of TLS_{PS}**, but it remains to be seen whether the vendors are willing to deploy such a dynamic negotiation logic on their proxies.

5.5 Limitations

During our experiments in this research, we faced a few technical challenges. We could not locate IP addresses of the proxy servers for Opera Mini, Opera Mini Beta, Opera, Opera Beta, and UC Mini automatically using our heuristic-based approach, since the network addresses as well as the location of the proxy servers appears to be completely different in the browser-side and server-side traffic. Even the volume of traffic in both directions cannot be easily matched, since these browsers compress data in TLS_{CP} which reduces the volume significantly. Therefore, we had to resort to manual analysis for those traces, and tried to fully or partially match the **organization name** field from the locations of the proxy servers to determine the addresses of proxy servers.

As discussed in Sect. 3.4, another limitation of our experiments is that unlike previous work [11,37], we cannot install a custom CA certificate on the proxy servers of TI class browsers, and we found that *none of the HT class browsers trust the Android system CA store*. Consequently, we were not able to perform a fine-grained analysis of their certificate validation procedures, as we have to resort to obtaining certificates from commercial CAs, and they are quite restrictive in what to issue (Sect. 4.1).

6 Related Work

TLS interception and its effects on security was studied before, where Xavier et al. [11] designed a framework to test TLS proxies used in some antivirus and parental control software, Waked et al. [37] developed a framework for analyzing TLS interception performed by enterprise-grade network appliances, and Zakir et al. [14] presented a comprehensive study on the prevalence and security impact of HTTPS interception made by middleboxes and antivirus software. Previous research has also identified TLS intercepting antivirus and content filtering software are the main contributors of forged certificates [17]. These studies have shown that many TLS intercepting products are negatively impacting security, and their proxy implementations are often problematic.

This paper makes new contributions in two directions. First, we note that proxy-based mobile browsers is another class of appliances that performs TLS interception but not studied before by previous work, and second we include in our study browsers that use an alternative model of HTTP tunneling, which comes with its own security trade-offs and considerations.

Orthogonal to this line of research, researchers have studied the affects of TLS vulnerabilities on Web security [10]. Moreover, there have been studies on whether general Web security mechanisms (e.g., HSTS, CSP, Referrer Policy, etc.) are being supported by mobile browsers [21], and some Android banking apps were also found to have weaknesses regarding certificate validation [27], along with other issues in the choice of cipher suites, signature algorithms and TLS versions, which resonate greatly with our findings presented in this paper.

7 Conclusion

In this paper, we explore the security implications of proxy-based mobile browsers on Android, and found that many of these browsers do not provide adequate security guarantees to their users. Problems include the willingness to support weak ciphers and insecure TLS versions, as well as unwarranted leniency in certificate validation, which can open door to a variety of attacks. In many cases, the proxies' transparent leniency towards subpar TLS connections with the remote server and resulting asymmetry in strength of TLS parameters could potentially lead to a false sense of security. Apart from reducing bandwidth consumption, part of the reason why proxy-based browsers are gaining popularity is their supposed ability to protect user privacy and circumvent censorship. However, the findings of our study suggest that users should be cautions and make informed decisions on which browser to use, or risk serious repercussions.

Acknowledgement. We thank the anonymous reviewers for their comments. This work is supported by NSF grant CNS-1657124.

References

1. Baseline requirements for the issuance and management of publicly-trusted certificates (2019). https://cabforum.org/wp-content/uploads/CA-Browser-Forum-BR-1.6.6.pdf
2. Al Fardan, N.J., Paterson, K.G.: Lucky thirteen: breaking the TLS and DTLS record protocols. In: IEEE S&P (2013)
3. AlFardan, N., Bernstein, D.J., Paterson, K.G., Poettering, B., Schuldt, J.C.: On the security of RC4 in TLS. In: USENIX Security (2013)
4. Barker, E., Mouha, N.: Recommendation for triple data encryption algorithm (TDEA) block cipher. NIST special publication 800–67 Rev. 2 (2017)
5. Barker, E., Roginsk, A.: Transitioning the use of cryptographic algorithms and key lengths. NIST special publication 800–131A Rev. 2 (2019)
6. Bellare, M.: New proofs for NMAC and HMAC: security without collision-resistance. In: Annual International Cryptology Conference (2006)
7. Bhargavan, K., Leurent, G.: On the practical (in-) security of 64-bit block ciphers: collision attacks on http over TLS and openVPN. In: ACM CCS (2016)
8. Bhargavan, K., Leurent, G.: Transcript collision attacks: breaking authentication in TLS, IKE, and SSH. In: NDSS (2016)
9. Bright, P.: Apple, google, microsoft, and mozilla come together to end TLS 1.0 (2018). https://arstechnica.com/gadgets/2018/10/browser-vendors-unite-to-end-support-for-20-year-old-tls-1-0/
10. Calzavara, S., Focardi, R., Nemec, M., Rabitti, A., Squarcina, M.: Postcards from the post-http world: amplification of https vulnerabilities in the web ecosystem. In: IEEE S&P (2019)
11. de Carnavalet, X.D.C., Mannan, M.: Killed by proxy: analyzing client-end TLS interception software. In: NDSS (2016)
12. Debnath, J.: When TLS meets proxy on mobile (2020). https://sites.google.com/view/joyantadebnath/when-tls-meets-proxy-on-mobile
13. Duong, T., Rizzo, J.: Here come the $\oplus$ ninjas. Technical report (2011)

14. Durumeric, Z., et al.: The security impact of https interception. In: NDSS (2017)
15. Ensafi, R., Fifield, D., Winter, P., Feamster, N., Weaver, N., Paxson, V.: Examining how the great firewall discovers hidden circumvention servers. In: ACM IMC (2015)
16. Garman, C., Paterson, K.G., Van der Merwe, T.: Attacks only get better: password recovery attacks against RC4 in TLS. In: USENIX Security (2015)
17. Huang, L.S., Rice, A., Ellingsen, E., Jackson, C.: Analyzing forged SSL certificates in the wild. In: IEEE S&P (2014)
18. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Lucky 13 strikes back. In: ACM Symposium on Information, Computer and Communications Security (2015)
19. Jarmoc, J., Unit, D.: SSL/TLS interception proxies and transitive trust. Black Hat Europe (2012)
20. Lenstra, A., De Weger, B.: On the possibility of constructing meaningful hash collisions for public keys. In: Australasian Conference on Information Security and Privacy (2005)
21. Luo, M., Laperdrix, P., Honarmand, N., Nikiforakis, N.: Time does not heal all wounds: a longitudinal analysis of security-mechanism support in mobile browsers. In: NDSS (2019)
22. McDonald, A., et al.: 403 forbidden: a global view of CDN geoblocking. In: ACM IMC (2018)
23. Moriarty, K., Farrell, S.: Deprecating TLSV1.0 and TLSV1.1 (2019). https://tools.ietf.org/html/draft-ietf-tls-oldversions-deprecate-05
24. Niaki, A.A., et al.: IClab: a global, longitudinal internet censorship measurement platform. In: IEEE S&P (2019)
25. Payne, B.: PKI at scale using short-lived certificates. In: USENIX Enigma (2016)
26. PCI Security Standards Council: Migrating from SSL and early TLS. Technical report (2015)
27. Reaves, B., et al.: Mo (bile) money, mo (bile) problems: analysis of branchless banking applications. ACM Trans. Priv. Secur. **20**(3), 1–31 (2017)
28. Ronen, E., Paterson, K.G., Shamir, A.: Pseudo constant time implementations of TLS are only pseudo secure. In: ACM CCS (2018)
29. Sotirov, A., et al.: MD5 considered harmful today, creating a rogue CA certificate. In: Annual Chaos Communication Congress (2008)
30. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full sha-1. In: Annual International Cryptology Conference (2017)
31. Stevens, M., Karpman, P., Peyrin, T.: Freestart collision for full sha-1. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques (2016)
32. Stevens, M., Lenstra, A., Weger, B.: Chosen-prefix collisions for md5 and colliding x.509 certificates for different identities. In: Annual International Conference on Advances in Cryptology (2007)
33. Taylor, M.: Tls 1.0 and 1.1 removal update (2019). https://hacks.mozilla.org/2019/05/tls-1-0-and-1-1-removal-update/
34. Topalovic, E., Saeta, B., Huang, L.S., Jackson, C., Boneh, D.: Towards short-lived certificates. Web 2.0 Secur. Priv. (2012)
35. VanderSloot, B., McDonald, A., Scott, W., Halderman, J.A., Ensafi, R.: Quack: scalable remote measurement of application-layer censorship. In: USENIX Security (2018)
36. Vanhoef, M., Piessens, F.: All your biases belong to us: Breaking RC4 in wpa-tkip and TLS. In: USENIX Security (2015)
37. Waked, L., Mannan, M., Youssef, A.: To intercept or not to intercept: analyzing TLS interception in network appliances. In: ACM AsiaCCS (2018)

38. Wang, Q., Gong, X., Nguyen, G.T., Houmansadr, A., Borisov, N.: Censorspoofer: asymmetric communication using IP spoofing for censorship-resistant web browsing. In: ACM CCS (2012)
39. Wang, X., Feng, D., Lai, X., Yu, H.: Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. IACR Cryptology ePrint Archive (2004)
40. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Annual International Cryptology Conference (2005)
41. Wilson, K.: Phasing out certificates with 1024-bit RSA keys (2014). https://blog.mozilla.org/security/2014/09/08/phasing-out-certificates-with-1024-bit-rsa-keys/