An Experimental Study of TLS Forward Secrecy Deployments

Lin-Shung Huang*, Shrikant Adhikarla†, Dan Boneh‡, Collin Jackson*
*Carnegie Mellon University, {linshung.huang, collin.jackson}@sv.cmu.edu
†Microsoft, shrikant.adhikarla@gmail.com
‡Stanford University, dabo@cs.stanford.edu

***Abstract*—Forward secrecy guarantees that eavesdroppers simply cannot reveal secret data of past communications. While many TLS servers have deployed the ephemeral Diffie-Hellman (DHE) key exchange to support forward secrecy, most sites use weak DH parameters resulting in a false sense of security. In our study, we surveyed a total of 473,802 TLS servers and found that 82.9% of the DHE-enabled servers were using weak DH parameters. Furthermore, given current parameter and algorithm choices, we show that the traditional performance argument against forward secrecy is no longer true. We compared the server throughput of various TLS setups, and measured real-world client-side latencies using an ad network. Our results indicate that forward secrecy is no harder, and can even be faster using elliptic curve cryptography (ECC), than no forward secrecy. We suggest that sites should migrate to ECC-based forward secrecy for both security and performance reasons.**

***Keywords*-TLS; forward secrecy; elliptic curve cryptography;**

I. Introduction

The Transport Layer Security (TLS) protocol is designed to provide privacy and data integrity between two communicating parties. TLS is increasingly used on the Internet to protect users' emails, credit card transactions and other personal data. We briefly describe how TLS protects data from eavesdroppers as follows. When a client connects to a TLS server, the client generates a random nonce (the *pre-master secret*), encrypts it with the server's public key, and sends it to the server. The pre-master secret is used by both parties to derive a shared session key for bulk encryption. Since the pre-master secret is encrypted with the server's public key, only the holder of the server's private key should be able to decrypt encrypted messages. However, the security of the server's private key is not always as robust as one may wish. An attacker could possibly steal private keys from server administrators via social engineering, recover expired private keys from discarded storage devices (that might be less protected), or perform cryptanalysis with future super computers. In fact, the Heartbleed OpenSSL bug [1] makes a point that private keys can be silently stolen from servers. Hence, an eavesdropper capturing encrypted traffic today might be able to decrypt the traffic in the future. A leaked document [2] even suggests that some governments have surveillance programs to capture backbone communications. If a government stores all of the captured traffic and later requests the server's private key, then past encrypted communications may be decrypted.

In response, some sites like Google have deployed forward secrecy to protect the privacy of past encrypted communications [3], using TLS's ephemeral Diffie-Hellman (DHE) or ephemeral Elliptic Curve Diffie-Hellman (ECDHE) key exchange methods. With these methods, the server's long-term secret key is used to sign a short-lived (ephemeral) Diffie-Hellman key exchange message. The resulting Diffie-Hellman secret is used as the session's pre-master secret. Once the pre-master secret is discarded after the session, the session key cannot be reconstructed even if the server's private key is given. Unfortunately, we discovered in our study that 82.9% of web sites who support DHE are using DH parameters weaker than their signature key strengths, resulting in a weakened pre-master secret. This suggests an industry-wide misunderstanding of how DHE should be deployed.

Our results. In this study, we performed a survey of TLS forward secrecy deployment across the top one million websites. We then evaluate the performance costs of forward secrecy on servers and real-world clients. Some of our results are as follows:

We successfully scanned 473,802 websites that use TLS. Over 74% of those sites supported at least one of the two ephemeral key exchange methods (DHE or ECDHE). Unfortunately, we discovered that 82.9% of the servers supporting DHE used Diffie-Hellman parameters that are weaker than their RSA signature strengths. The resulting sessions are therefore more vulnerable to a brute-force cryptanalysis attack than if the RSA key exchange (with no forward secrecy) was used.

We evaluated the throughput of TLS servers and find that ECC-based forward secrecy is not much slower, and can even be faster, than RSA-based setups with no forward secrecy. The reason is that with the RSA key exchange, the server must perform an expensive RSA decryption on *every* key exchange. With the ECDHE key exchange, the server can RSA-sign its ECDH parameters once and re-use that signature across several connections. The server-side online cryptographic operation is then just one elliptic curve multiplication which can be faster than a single RSA decryption (of equivalent key strength). Consequently, the traditional performance argument against forward-secrecy is simply no longer valid given the current parameter and algorithm choices. Our results suggest that websites should move to using ECDHE and reap the security *and* performance benefits.

Lastly, we measured TLS client latencies on thousands of real-world clients using ad networks. We identified a one-time performance issue that occurs when a client first sees an ECDSA signature. Encouragingly, our TLS latency measurements indicate that ECC-based forward secrecy performs no worse than plain RSA-based setups on the real-world clients.

Organization. The rest of this paper is organized as follows. Section II provides background and related work. Section III presents a survey of TLS servers, which reveals that most DHE servers use insufficient key strengths. Section IV evaluates the server and client performance of common TLS cipher suites that support forward secrecy. Section V discusses the best practices for deploying forward secrecy. Section VI concludes.

II. Background

In this section, we give a brief introduction to the TLS protocol and forward secrecy. We then survey related work.

A. *SSL/TLS*

Transport Layer Security (TLS) [4], the successor of Secure Sockets Layer (SSL) [5], is an encryption protocol designed to prevent network adversaries from eavesdropping or tampering sensitive data, while enabling clients and servers to reliably identify each other. TLS allows application-specific protocols such as HTTP to be encapsulated and securely transmitted over the underlying transport protocols such as TCP, and is increasingly used by web applications, especially by webmail and online banking websites.

The TLS protocol is designed to support an extensible set of *cipher suites*, where each cipher suite defines a combination of authentication, key exchange, bulk encryption, and message authentication code (MAC) algorithms to be used.

- *Authentication* algorithms allow communicating parties to verify the identities of each other, e.g., RSA and DSA.
- *Key exchange* schemes allow peers to securely agree upon on a session key used for bulk encryption, e.g., RSA and Diffie-Hellman.
- *Bulk encryption* ciphers are used to encrypt the application data, e.g., AES and RC4.
- *Message authentication* algorithms are used to generate message digests, e.g., SHA-1.

Recent versions of TLS now support elliptic curve cryptography (ECC) [6], including ECDSA authentication and Elliptic Curve Diffie-Hellman key exchange.

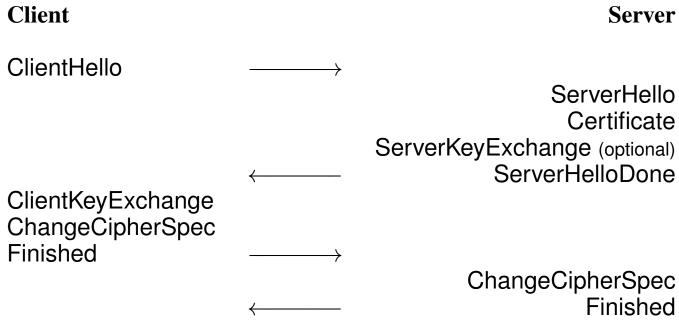


Fig. 1: A TLS handshake with no client authentication. The optional ServerKeyExchange message is sent when using the Diffie-Hellman key exchange.

When initializing a TLS connection, the client negotiates the cipher suite and parameters with the server over a TLS handshake. Figure 1 depicts a standard TLS handshake using the basic RSA key exchange with no client certificates. Initially, the client sends a ClientHello message to the server to provide a list of supported cipher suites (in preference order) and a client random nonce. In response, the server sends a ServerHello message that specifies the chosen cipher suite and a server random nonce. Note that the server determines the cipher suite and may ultimately override the client's preference order. The server also sends its public key certificate over the Certificate message to the client. Since the message may be manipulated by a network attacker, the client is responsible of checking whether the server's certificate is valid (e.g., issued by a trusted certificate authority, matches the server's hostname, not expired nor revoked, etc.). If the certificate is accepted, the client generates a pre-master secret, encrypts this pre-master secret using the server's public key, and sends the encrypted secret to the server over a ClientKeyExchange message. At this point, both the client and server can compute a common master secret (used to derive the session key) from the pre-master secret and the random nonces. Lastly, both the server and the client send ChangeCipherSpec messages, indicating that subsequent messages will be encrypted using the negotiated cipher suite and session key, starting with an encrypted Finished message.

B. *Forward secrecy*

Forward secrecy, also known as perfect forward secrecy (PFS), is an important security property which guarantees that derived session keys cannot be revealed, even if the long-term private key is compromised in the future. Especially in the situation where Internet surveillance is a concern, forward secrecy lets enterprises argue that eavesdroppers simply cannot reveal secret data of past communications. However, in TLS, forward secrecy is not necessarily guaranteed. In particular, the RSA key exchange is only secure as long as the server can protect its private key. If the server's private key is ever revealed, an attacker can decrypt all recorded sessions by deriving the pre-master secret using the server's private key, and basically recover all past session keys.

There are currently two key exchange methods in TLS that support forward secrecy, including ephemeral Diffie-Hellman (DHE) and ephemeral Elliptic Curve Diffie-Hellman (ECDHE). When using DHE or ECDHE, the server's long-term secret key is used to sign a short-lived Diffie-Hellman key exchange message as the pre-master secret (that is discarded after the session). For example, when using DHE key exchange with RSA signatures, the server sends an additional ServerKeyExchange message which contains an ephemeral Diffie-Hellman public key that is signed with server's RSA private key. Similarly, when using ECDHE with RSA signatures, an extra ServerKeyExchange message contains the ephemeral elliptic curve Diffie-Hellman public key and its elliptic curve domain parameters, which are signed with the server's RSA private key. The server may also replace RSA signatures entirely with elliptic curve cryptography, by signing the ECDHE public key with its ECDSA private key.

C. Related work

Several past studies [7], [8], [9], [10] have crawled public websites to measure the prevalence of specific server vulnerabilities in the wild. For instance, Lee et al. [9] scanned 19,429 TLS servers checking support for weak export ciphers and insecure protocol versions. The SSL Pulse project [11] periodically scans the top TLS servers for known vulnerabilities such as insecure renegotiation, BEAST and CRIME. Another category of TLS surveys [12], [13], [14], [15] analyzed TLS certificates of public websites to measure common server misconfigurations. For instance, Holz et al. [13] scanned the Alexa top one million sites and monitored TLS traffic of a research network over a 1.5 year period, reporting that roughly 40% of the observed certificates failed to validate. In comparison, our TLS survey focuses on studying forward security deployments and identifying insecure DHE or ECDHE configurations.

A number of TLS performance evaluations [16], [17], [18], [19], [20] have been conducted in the past. Coarfa et al. [17] profiled TLS web servers with trace-driven workloads in 2002, showing that the largest performance cost on the TLS web server is from the RSA operations, and suggested that TLS overhead will diminish as CPUs become faster. Gupta et al. [18] showed that TLS server throughput can increase by 11% to 31% when using ECDSA signatures over RSA signatures. Bernat [19] evaluated RSA, DHE and ECDHE key exchanges over 1,000 handshakes and reported a 15% server overhead for using ECDHE-256 over RSA-2048 key exchange. Subsequently, Mavrogiannopoulos [20] reported that at equivalent security levels, ECDHE-192 outperforms RSA-1776 key exchange. In our work, we are first to conduct an ad experiment to measure client perceived TLS latencies in real-world.

III. Survey of TLS servers

We conducted our survey to find out how many servers currently support forward secrecy, and whether servers use secure Diffie-Hellman parameters. We describe our methodology for scanning websites first and then present our results.

A. Methodology

We surveyed the top 1,000,000 global sites (retrieved from Alexa [21] on September 9, 2013) during September 13, 2013 to September 27, 2013. The Alexa dataset is a ranked list of the most popular sites on the Internet, and has been widely used by several TLS server surveys [9], [13], [15]. Since Alexa's rankings are not based on TLS traffic, many of the listed sites do not actually support TLS, thus we used a simple heuristic to discover TLS-enabled servers. For each site in the Alexa list, we tried connecting to port 443 of the naked domain, and then the www subdomain (if the former failed).

Error	Hosts
Connection refused error	163,948
SSL errors	187,532
Timeout	120,068
Invalid hostname error	41,209
Connection reset error	11,915
IP unreachable	1,325
DoS-blocked	192
Other errors	9

TABLE I: Connection errors during TLS survey

We implemented our scanner based on SSLScan 1.8.2 [22]. By making multiple TLS connections to the host, SSLScan determines the supported SSL/TLS protocol versions, supported cipher suites, and cipher suite preference of a TLS server. We updated the code to test all of the current TLS cipher suites available in OpenSSL 1.0.1e [23], and modified our OpenSSL to expose TLS key exchange details, including the actual DH parameters and ECDH parameters from the servers. Since the scanning process is I/O bound due to network latencies, we implemented a Python script that spawns a pool of 300 concurrent scanner processes, such that multiple sites are scanned in parallel (rather than in serial). In addition, we set a socket connection timeout of 30 seconds.

In our study, we made no attempt to validate the TLS certificates, thus our dataset may include reused certificates on shared hosts, and possibly some configurations of servers not actually in use. This particular subject has been investigated by several existing studies [13], [14], [15], [12].

B. Results

We successfully performed our TLS scan on 473,802 unique hosts out of the top one million sites. Note that distinct hostnames may share the same IP or physical machine. Table I summarizes the connection errors of the 526,198 Alexa entries where both the naked domain and the www subdomain failed. Most of connection failures (e.g. timeout, connection refused, connection reset) were simply because websites did not run a TLS server on port 443. The SSL errors were possibly caused by server misconfigurations, which triggered TLS alert messages before the TLS handshake has completed. Most of the invalid hostname errors were caused by hostname strings in the Alexa dataset that incorrectly contained a URL path component (which otherwise overlaps with valid hostnames in the dataset).

There were 192 errors that we labeled as "DoS-blocked", which means the site actually accepted our initial TLS connection, but stopped responding to our machine before we were able to test all of the available cipher suites. We reason that these websites deployed defenses against denial-of-service (DoS) attacks, which blocked our IP address after making

Method	Hosts	IMC'07[9]
RSA	473,688 (99.9%)	99.86%
DHE	283,647 (59.8%)	57.57%
ECDHE	85,070 (17.9%)	
Fixed ECDH	1 (0.0%)	

TABLE II: Key exchange method support on TLS servers

Size (bits)	Hosts
256	2 (0.0%)
512	96,559 (34.0%)
768	933 (0.3%)
1024	281714 (99.3%)
1544	1 (0.0%)
2048	859 (0.3%)
3248	2 (0.0%)
4096	14 (0.0%)

TABLE III: Diffie-Hellman parameter size support for DHE key exchange

consecutive connections in a short period of time. While there may be workarounds to this (e.g., acquiring more IP addresses to open connections from, or lowering our connection rate), the number of sites affected was not significant. We present the main results of the successful TLS scans below.[1]

1) Key exchange methods: First of all, we show in Table II the key exchange methods supported by websites, comparing to Lee et al.'s study in 2006 [9]. Since each unique host may use multiple SSL/TLS protocol versions, we consider that a particular key exchange method is supported by a host if it worked in at least one of the SSL/TLS protocol versions in use. In our results, RSA is clearly the most widely supported key exchange method, accepted by over 99.9% TLS sites. There were 59.8% TLS sites that supported DHE (which barely increased from 57.5% since 2006). However, only 17.9% of the TLS sites actually supported ECDHE. One possible reason for the low ECDHE adoption rate is that elliptic curve cryptography has been intentionally disabled in popular Linux distributions (Red Hat / Fedora) due to patent concerns until 2013 [24].

As mentioned in Section II-B, both the DHE and ECDHE key exchanges support forward secrecy while the RSA key exchange does not. In all, there were a total of 353,209 (74.5%) unique hosts that enabled either DHE or ECDHE, thus supported forward secrecy. Out of the servers that supported forward secrecy, we noticed that 287,301 (81%) of these servers preferred to use either DHE or ECDHE over RSA.

Weak ephemeral DH parameters. In Table III, we show the actual Diffie-Hellman parameter sizes supported by TLS servers that enabled the DHE key exchange. We note that a single host may support multiple DH parameter sizes, thus adding up the percentages of supporting hosts across different DH parameter sizes may exceed 100%. To our surprise, only 0.3% of the DHE-enabled servers have actually deployed 2048-bit DH parameters! Our results show that as much as 99.3% of the DHE-enabled servers supported 1024-bit DH parameters. This is concerning since CAs and browsers are moving to 2048-bit (or stronger) RSA authentication by 2014 [25], and 1024-bit Diffie-Hellman may soon be considered insufficient. Even worse, there were 34% DHE-enabled servers that deployed insecure 512-bit DH parameters, mostly due to supporting export cipher suites.

Curve	Hosts
secp256r1	81,789 (96.1%)
sect233r1	3,123 (3.6%)
sect571r1	316 (0.3%)
secp384r1	86 (0.1%)
secp521r1	73 (0.0%)
sect163r2	26 (0.0%)
secp224r1	3 (0.0%)
secp192r1	1 (0.0%)

TABLE IV: Elliptic curves used for ECDHE key exchange

We further examine how many TLS servers are using DHE with an ephemeral DH parameter size smaller (weaker) than its RSA/DSA key size. Unfortunately, the majority (82.9%) of the DHE-enabled servers supplied smaller DH parameters. On 61.9% of the DHE-enabled servers, the server used smaller DH parameters and used DHE as server's preferred key exchange. In other words, if the browser supports both RSA and DHE, the server would use the weakened DHE by default.

To explain the widespread usage of weak DH parameters, we cite the fact that popular web servers simply did not support 2048-bit DHE out of the box (until recently) [26], [27]. For example, Apache (prior version 2.4.7) only supported up to 1024-bit DH parameters [28], and IIS uses 768-bit DH parameters [29]. Interestingly, Graham [30] pointed out that Tor (prior version 2.4) uses 1024-bit DHE as well. Another possible pushback against increasing DHE strength on servers is that Java clients only support up to 1024-bit DH parameters. Fortunately, a recent patch has lifted this limitation [31].

Consistent ephemeral ECDH parameters Encouragingly, out of all the successful ECDHE connections, none of the ephemeral ECDHE keys were weaker than the RSA signatures in use. This may be explained by the fact that the key strength of the most commonly used 256-bit ECDHE is stronger than the currently recommended 2048-bit RSA signature.

Table IV lists the elliptic curves observed over the ECDHE key exchange during our TLS survey when connecting with an OpenSSL client. Most of the ECDHE-enabled sites used the curve named secp256r1, also known as NIST P-256 [32]. One should expect similar results using browsers, since Firefox and Google Chrome only support three NIST curves

[1] Additional data (SSL/TLS versions, encryption methods, and MAC methods) from our TLS scan are provided in Appendix A.

Method	Hosts	IMC'07[9]
RSA	473,780 (99.9%)	≥ 99.86%
Anonymous	7,750 (1.6%)	
DSA	22 (0.0%)	0.02%
ECDSA	2 (0.0%)	
ECDH	1 (0.0%)	

TABLE V: Authentication method support on TLS servers

Size (bits)	Hosts	IMC'13[15]	IMC'07[9]
≤ 512	350 (0.0%)	0.1%	3.94%
513 - 1023	20 (0.0%)	0.0%	1.42%
1024	87,760 (18.5%)	10.5%	88.35%
1025 - 2047	20 (0.0%)	0.7%	0.01%
2048	374,294 (79.0%)	86.4%	6.14%
2049 - 4095	251 (0.0%)	0.0%	0.00%
4096	11,093 (2.3%)	2.3%	0.19%
≥ 4097	22 (0.0%)	0.0%	0.00%

TABLE VI: RSA key sizes of TLS server certificates

(secp256r1, secp384r1 and secp521r1), and Internet Explorer only supports two curves (secp256r1 and secp384r1).

During our scan, we noticed that 34,145 hosts (40.1% of the ECDHE-enabled servers) reused their ECDHE public keys for multiple connections, which means that these servers did not re-sign new ECDH parameters for every connection. This may be a deliberate performance optimization, as long as the server re-generates new parameters periodically.

2) Authentication methods: Table V lists the authentication methods that were supported by websites. As shown, RSA is by far the most commonly deployed authentication method. This result has not changed much since 2006 [9]. We only observed two hosts that supported ECDSA authentication. By manual inspection, we observed that one host used a self-signed ECC certificate, while the other host used a valid ECC certificate signed by Symantec. There was one odd host that used ECDH authentication with a self-signed certificate for the fixed ECDH key exchange. Unfortunately, we noticed there are 1.6% hosts that still support *anonymous* cipher suites, which offers no authentication at all and are trivially vulnerable to TLS man-in-the-middle attacks.

Next, we present the distribution of RSA public key sizes of TLS sites that use RSA signatures in Table VI. Each unique host may possibly support multiple public key sizes. Note that the percentages published by Durumeric et al. [15] are calculated differently; they represent the distribution of each public key size by counting unique trusted certificates, rather than unique hosts; furthermore, they excluded certificates that are not trusted by browsers, thus differences from our results can be expected. Nevertheless, we find our results roughly in line with their study. Usage of 1024-bit RSA public keys have significantly dropped since 2006 [9], while 2048-bit keys have increased to 79% of the measured TLS servers.

Cipher suite[a]	Key exchange	Authentication	PFS
RSA-RSA	RSA-2048	RSA-2048	No
DHE-RSA	DHE-2048	RSA-2048	Yes
ECDHE-RSA	ECDHE-256	RSA-2048	Yes
ECDHE-ECDSA	ECDHE-256	ECDSA-256	Yes
DHE-DSA	DHE-2048	DSA-2048	Yes

[a] For brevity, we abbreviated the actual TLS cipher suite names, e.g., TLS_DHE_RSA_WITH_AES_128_CBC_SHA as DHE-RSA.

TABLE VII: TLS cipher suites for evaluation

IV. TLS PERFORMANCE EVALUATION

In this section, we analyze the performance of various TLS cipher suites that support forward secrecy, on the server side and the client side. We conducted two separate experiments to evaluate the performance of various TLS cipher suites. First, we conducted a *controlled experiment* where we load tested our TLS servers over a high-speed internal network. Second, we ran an *Ad experiment* to measure the client-side TLS latencies on real-world clients.

A. TLS server setup

In our experiments, we used Apache 2.4.4 compiled with OpenSSL 1.0.1e (with 64-bit elliptic curve optimizations [33]) to run our TLS servers. Abalea's mod_ssl patch [28] was applied to support 2048-bit Diffie-Hellman parameters. [2] We used Rackspace virtual private servers (a generic low-end setup for running modern web servers) equipped with AMD Opteron 4170 HE 2.1 GHz CPU, 512 MB RAM and 40 Mbps network bandwidth, installed with Debian Linux 2.6.32-5. We setup multiple TLS virtual hosts on distinct ports, in which each TLS virtual host enabled only a single cipher suite.

We selected five representative TLS cipher suites for evaluation as summarized in Table VII. All of the cipher suites in our experiments were uniformly configured to use 128-bit AES-CBC encryption with SHA-1 HMAC. However, the security strengths of these cipher suites were not necessarily equivalent (e.g., ECDSA-256 is stronger than RSA-2048 and DSA-2048). This is mainly because our server certificates were issued by a commercial CA, which does not allow configuring arbitrary certificate strengths. Note that we could not simply use self-signed certificates, since they trigger SSL certificate warnings on real-world clients in our ad experiment. Table VIII compares the 3 production TLS certificate chains used in our evaluation, listing the signature algorithms, signature hash algorithms and chain sizes. Different signature algorithms (RSA, DSA and ECDSA) were not mix-and-matched within the same chain. We point out that there is roughly an one kilobyte size difference between the RSA and ECDSA certificate chains. This is because the a ECDSA-256 public key plus signature is smaller than a RSA-2048 public key plus signature, and there are two certificates (leaf and intermediate) transmitted per chain. Lastly, since none of the major browsers currently

[2] Abalea's patch is obsolete as of Apache 2.4.7 [34].

	Leaf certificate	Intermediate certificate	Root certificate (not transmitted)	Chain size (bytes)
1.	RSA-2048, SHA-256	RSA-2048, SHA-1	RSA-2048, SHA-1	3,119
2.	DSA-2048, SHA-256	DSA-2048, SHA-256	DSA-2048, SHA-256	3,343
3.	ECDSA-256, SHA-256	ECDSA-256, SHA-384	ECDSA-384, SHA-384	2,104

TABLE VIII: TLS certificate chains issued by Symantec CA for evaluation

support DSA-2048 signatures (despite that some support DSA-1024), DHE-DSA was not measured in the ad experiment.

B. Controlled experiment

1) Methodology: In the controlled experiment, we measured the average server throughput of each TLS server setup by generating large amounts of synthetic TLS traffic towards the server, from two client machines over a 40 Mbps private network. We used the ApacheBench tool [35] to send HTTPS requests continuously, and concurrently (1,000 requests at the same time), from each client machine. We disabled TLS session resumption and HTTP connection re-use. We monitored the server throughput (number of requests per second) and took the average value over 5 minutes. For sanity check, we tested each TLS server configuration using GET requests and HEAD requests, separately.

Since the performance of TLS servers may possibly vary by the complexity of web pages, we setup three different types of web pages for our experiments, all mirrored from real websites on the Internet:papp

- Simple page - a copy of one of our author's home page. The page was static and hosted on a single domain.
- Complex page - a copy of Amazon.com's landing page. We downloaded the page along with its sub-resources (e.g. images, stylesheets and scripts), and hosted the page and all sub-resources on a single domain.
- Multi-domain page - a copy of Salon.com's landing page. After downloading the page, we manually categorized its sub-resources according to their originating domains. We then setup 10 sub-domains on our site (corresponding to the originating domains) to serve the sub-resources, and modified the landing page to request its sub-resources from those sub-domains.

2) Results: In Figure 2, we show the average number of requests per second that the web server can serve when fully loaded under each server configuration. First, we observe the impact of key exchange schemes on server throughput by comparing the three cipher suites (RSA-RSA, DHE-RSA and ECDHE-RSA) that use the same signature algorithm (RSA-2048) but different key exchange schemes (RSA, DHE and ECDHE). Our results show that RSA-RSA was clearly the fastest of the three regardless of the type of web page, peaking at 265.4 GET requests per second (when serving simple pages). Notably, DHE-RSA performed significantly slower than RSA-RSA and ECDHE-RSA, averaging 45.7 requests per second in the best case. This should be due to the extra computation required for generating the ephemeral DH key (and RSA-signing it) for each ServerKeyExchange message. Interestingly, ECDHE-RSA averaged 237 requests per second (when serving simple pages), suggesting that the server performance cost of forward secrecy using ECDHE is not only dramatically cheaper than DHE, but also almost free compared to plain RSA.

Next, we look at how different signature algorithms impact server throughput. By comparing DHE-RSA and DHE-DSA, we check whether RSA and DSA signature algorithms perform differently (using the same DHE key exchange). DHE-RSA averaged as low as 44.9 requests per second (when serving multi-domain pages). Similarly, DHE-DSA averaged as low as 44.8 requests per second (when serving simple pages). Evidently, the performance of DHE key exchange was consistently the worst regardless of using either RSA or DSA signatures. When using ECDSA signatures (matched with the ECDHE key exchange), we can see that the performance of ECDHE-ECDSA is the fastest (peaking at 405 requests per second when serving simple pages). ECDHE-ECDSA is not only faster than ECDHE-RSA (also using ECDHE key exchange), but even faster than RSA-RSA, which does not provide forward secrecy. Moreover, ECDSA-256 has a higher security strength than RSA-2048, thus one could expect a larger difference if comparing equivalent strengths. Most interestingly, this suggests that enabling forward secrecy may even improve server performance.

In order to reaffirm the observations above (for GET requests), we compare the corresponding measurements using HEAD requests, which skips transmitting the entire page body after the TLS handshake. We find that both results were consistent. We draw the following three main observations from our controlled experiment:

- DHE-RSA and DHE-DSA performed the slowest. *Using DHE, forward secrecy is very slow.*
- ECDHE-RSA is not far worse than RSA-RSA. *Using ECDHE, forward secrecy is basically free.*
- ECDHE-ECDSA performed faster than RSA-RSA. *Using elliptic curve cryptography, enabling forward secrecy actually improves performance.*

C. Ad experiment

1) Methodology: To compare the client-side performance of different TLS cipher suites, we conducted an ad experiment to measure TLS latencies on real-world clients. Ad networks have been used by researchers as a platform for measuring browser and network characteristics, such as finding security vulnerabilities [36], [37]. In this study, we served our perfor-

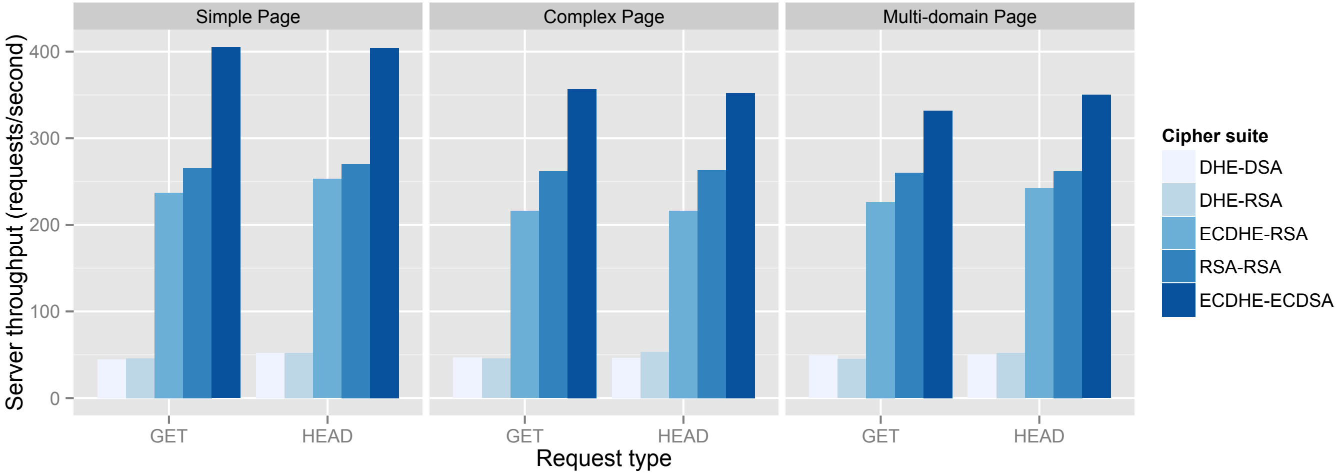


Fig. 2: Server throughput of different configurations under synthetic traffic

mance measurement code to thousands of browsers across the world over an ad network.

Our experiment setup consisted of two machines (with separate IP addresses and domain names), one which runs the TLS servers to be tested (as described in Section IV-A), and the other which hosts our advertisement banner page (mainly a blank image and JavaScript code). When our ad banner is rendered, our code will open TLS connections from the client to our TLS server by loading each HTTPS test link in an IFRAME. For each HTTPS test, our script measures the client-side latencies using the HTML5 Navigation Timing API [38] (on supporting browsers). We collected the following client-side latency measurements:

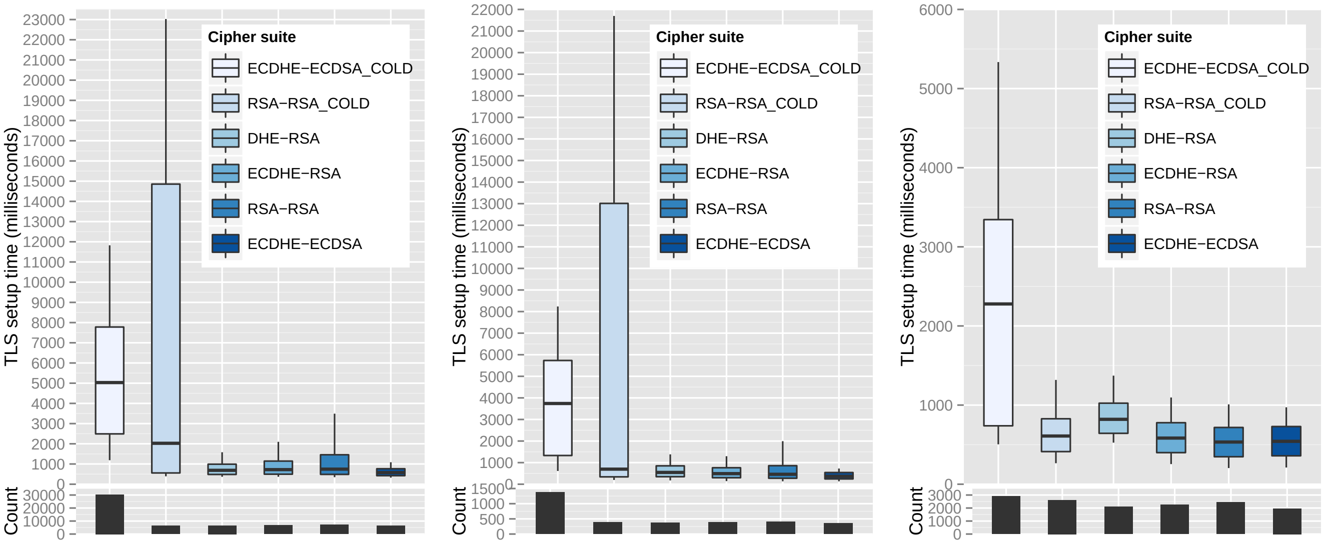
- *TLS setup time*: The amount of time used to establish a SSL/TLS connection, including the TLS handshake time and the certificate validation time on the client. Unfortunately, this measurement is only available in Chrome browsers (using Navigation Timing's optional `secureConnectionStart` attribute).
- *TCP + TLS setup time*: The amount of time used to establish the transport connection, which includes the TCP handshake and TLS handshake (and optionally SOCKS authentication). The required APIs (`connectEnd` and `connectStart`) are currently supported in three major browsers (including Chrome 6+, Firefox 7+ and IE 9+).

Upon completion, the timing measurements are sent back via a GET request to our log server. When receiving network requests, our log server immediately discards the client's IP address (to avoid storing information that might individually identify the viewer of our advertisement). The experiment did not require any user involvement. If the user navigated away (e.g. closed the tab) during the experiment, or if the TLS connection failed, our servers still received partial results.

As mentioned in Section IV-A, all HTTPS tests used separate server ports with TLS session resumption disabled. The test pages were loaded in serial on the client to reduce the interference in between each test. Since modern web clients are known to cache OCSP responses and we have three server setups that share the same RSA certificate chain, it is possible that whichever RSA-certificate setup tested first may load slower due to performing a fresh OCSP lookup, while subsequent tests may load faster due to enjoying a warmed OCSP cache. However, we cannot remotely flush the client's OCSP cache before each test, nor do we have the option of switching different intermediate certificates for our production certificates. As a workaround, we added "cold" TLS connections in front of our tests to warm the client's OCSP cache, such that all of the subsequent tests would be equally evaluated under a warmed OCSP cache.

2) Results: We purchased 273,533 advertisement impressions from 23 January 2014 to 29 January 2014. We spent $167.75 in total, including $122.23 on a run-of-network campaign (195,214 impressions), and $45.52 targeted on mobile devices (78,319 impressions). We note that not all ad impressions can convert to valid measurements. We discarded impressions with clients that do not support HTML5 Navigation Timing, and clients that are not viewing our ad for the first time. Also, users may leave the web page before completion of tests. We indicate the number of unique clients that successfully performed each test in Figures 3a-c and 4a-c. Since the measured client-side latencies may contain outliers, we visualize our results with a box plot showing the 10th, 25th, 50th (median), 75th and 90th percentiles for each setup.

TLS setup times. Figures 3a-c show the TLS setup times of different cipher suites in Chrome browsers on Windows, OS X and Android (note that other browsers do not support this measurement). When comparing client-side latencies, smaller values mean better performance (less waiting). As mentioned in Section IV-A, DHE-DSA was not measured since no major browser supported DSA-2048. Out of the four different cipher suites tested, two cipher suites were tested an additional time for the purpose of warming the OCSP cache, labeled as RSA-



(a) Chrome - Windows

(b) Chrome - OS X

(c) Chrome - Android

Fig. 3: Comparison of **TLS setup times** in Chrome browsers on Windows, OS X and Android. The box plots show the 10th, 25th, 50th, 75th and 90th percentiles of measured TLS setup times for each cipher suite. The corresponding bar charts show the number of unique clients that successfully completed each test.

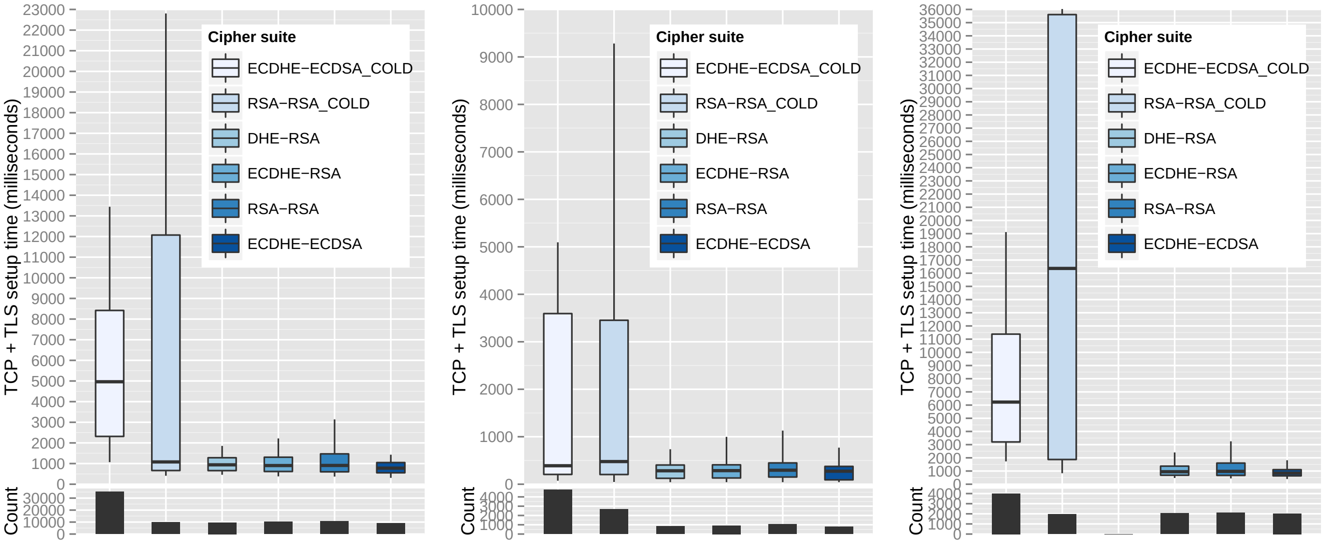
RSA_COLD and ECDHE-ECDSA_COLD. The number of unique clients for ECDHE-ECDSA_COLD appeared to be the highest because it was always tested first (and many users do not stay on the page long enough for other tests to complete).

Upon first glance at Figure 3a (Chrome on Windows), an obvious observation is that the medians of ECDHE-ECDSA_COLD and RSA-RSA_COLD are both substantially slower than the other configurations. Unsurprisingly, the two "cold" connections may need to perform OCSP lookups, resulting in longer latencies, while the subsequent tests may enjoy a warm OCSP cache. We compare DHE-RSA, ECDHE-RSA, RSA-RSA and ECDHE-ECDSA after warming the OCSP cache.

While the performances of the 4 different cipher suites (not labeled "cold") were not too dissimilar, we noticed that the ECDHE-ECDSA setup consistently performed the fastest of all setups, resulting in a median of 366 milliseconds (and a 90th percentile of 1088 milliseconds). This suggests that deploying ECC-based forward secrecy actually improves performance on the client over RSA-based setups with no forward secrecy. Encouragingly, we observe very similar trends on OS X (in Figure 3b) and Android (in Figure 3c), where the medians of TLS setup times for ECDHE-ECDSA were consistently the smallest. In particular, the measurements on Android provide an interesting data point showing that mobile devices (typically with less computational power than desktop clients) might also benefit from ECC-based forward secrecy. On the other hand, DHE-RSA performed the slowest on Android mobile with a median of 820 milliseconds (and a 10th percentile of 525 milliseconds). Moreover, DHE-RSA is not supported in one of the major browsers, Internet Explorer. Servers that want to support forward secrecy using RSA certificates should consider choosing ECDHE-RSA over DHE-RSA for both client compatibility and performance reasons.

TCP + TLS setup times. Since TLS setup time measurements were only available in Chrome browsers, on other browsers we fall back on collecting TCP + TLS setup time measurements. The coarser TCP + TLS setup time includes not only the TLS handshake but also the TCP handshake, thus may show a slightly longer delay (and possibly more noise incurred by the extra round-trips). We compare the TCP + TLS setup times of different cipher suites in Figures 4a-c for Chrome, Firefox and Internet Explorer browsers on all platforms. In Figure 4c, we do not have any results for DHE-RSA since it was not supported in Internet Explorer.

As a sanity check, the TCP + TLS results for Chrome in Figure 4a were basically in line with the TLS results in Figure 3a, where the ECDHE-ECDSA was the fastest of all cipher suites. For Firefox, we did not observe significant differences between the client latency medians for each cipher suite in Figure 4b. We did not find any cipher suite that performed particularly slower. Unlike Chrome (where ECDHE-ECDSA_COLD took roughly 4 seconds longer than other cipher suites in median), the performance of ECDHE-ECDSA_COLD in Firefox (a median of only 389 milliseconds) was not significantly slower than the fastest ECDHE-ECDSA (a median of only 274 milliseconds). Upon further investigation, we believe that "cold" connections in Firefox were often faster than other browsers because Firefox maintains its own root CA store (rather than rely on the operating system's root CA store). The underlying cause is that not all legitimate root CA certificates are pre-installed on popular client systems like Windows. When other browsers (using the system's CA root store) encounter an unseen root CA



(a) Chrome

(b) Firefox

(c) Internet Explorer

Fig. 4: Comparison of **TCP + TLS setup times** in Chrome, Firefox and Internet Explorer browsers (on all platforms).

certificate, the system automatically attempts to fetch the root certificate over the Windows Update mechanism [39]. In particular, we confirmed that Symantec's ECDSA root certificate was not pre-installed on Windows 8 or Windows Server 2012. As a result, the ECDHE-ECDSA_COLD setup performs slower in non-Firefox browsers on Windows because they may need to fetch the ECDSA root certificate on-the-fly when first seeing the ECDSA certificate.

Nevertheless, fetching a new root certificate is a one-time cost. As more TLS servers deploy ECDSA certificate chains, clients will eventually have downloaded the ECDSA root certificate after visiting any of those sites and will have payed off this one-time cost. From measurements on real-world clients, we discovered that without the root CA update, forward secrecy using ECDHE-ECDSA was not any slower than that using RSA-RSA.

V. Discussion

Forward secrecy deployment. Our experiments suggest that the performance-based arguments against deploying forward secrecy are no longer valid. ECDHE-based key exchange, which provides forward secrecy, can be faster than basic RSA-2048 key exchange which does not. The reason for the performance improvement is the replacement of an expensive RSA-2048 decryption with faster `secp256r1` elliptic curve operations. As we transition to longer RSA keys, such as RSA-3072 or RSA-4096, the performance advantage of ECDHE will become even more pronounced. These results suggest that sites should migrate to ECDHE (when possible) for both security and performance reasons.

The elliptic curve monoculture. We were a bit surprised to find in our study that 96.1% of sites that support ECDHE commonly use the NIST curve `secp256r1`. While we currently know of no specialized attacks against this curve, it is possible that a weakness with this specific curve will some day be discovered. Given the popularity of this curve, this will impact most sites on the Internet. Although `secp256r1` could be a fine curve to use for the foreseeable future, it is worth pointing out that we are putting many eggs in that curve's basket.

RSA vs. ECDSA authentication. Our survey shows that common practice today when using ECDHE is to use elliptic curves for key exchange, but use RSA signatures for server-side authentication. The reason is that sites mainly have certificates for RSA public keys. From a security standpoint this is an undesirable setup: a weakness discovered in *either* algorithm will defeat the security of TLS at the site. A-priori, the likelihood of a weakness discovered in one of two algorithms is far greater than the likelihood of an attack on a single algorithm. Consequently, due to the desire to move to ECDHE key exchange, there is a strong argument for sites to move to certificates for ECDSA public keys.

To understand the risk of using both RSA and ECDHE (called ECDHE-RSA) compared to only relying on elliptic curve cryptography (as in ECDHE-ECDSA), consider the following three possibilities:

1) both RSA and the NIST curve `secp256r1` provide adequate security,
2) curve `secp256r1` is secure, but RSA is not,
3) RSA is secure, but curve `secp256r1` is not.

Table IX lists the resulting security of ECDHE-RSA and ECDHE-ECDSA in each of the three cases. The table suggests that ECDHE-ECDSA incurs less risk than ECDHE-RSA since there is a scenario where ECDHE-ECDSA is secure, but ECDHE-RSA is not. The converse cannot happen. Given the desire to use ECDHE, Table IX is an argument for moving to elliptic curve public keys for server-side authentication.

To properly move to ECDSA signatures, CAs will need to

	ECDHE-RSA	ECDHE-ECDSA
RSA and `secp256r1` both secure	secure	secure
`secp256r1` secure, RSA insecure	**insecure**	**secure**
RSA secure, `secp256r1` insecure	insecure	insecure

TABLE IX: Comparing ECDHE-RSA and ECDHE-ECDSA

sign those certificates with ECDSA signatures along the entire certification chain. The security of TLS key exchange will then only depend on the hardness of a single algebraic problem instead of two. Only time will tell whether the elliptic curve discrete logarithm problem (on the NIST curve `secp256r1`) is indeed as hard as we currently believe.

Note that moving to ECDSA public keys means that during the ECDHE key exchange the server will need to generate an ECDSA signature. The ECDSA signature algorithm requires strong randomness: bias in the random generator can lead to exposure of the secret signing key [40]. Therefore, when moving to ECDSA public keys servers will need to ensure an adequate source of randomness. An alternative proposal, which is not frequently used, is to derive the ECDSA randomness by applying a PRF such as HMAC to the message to be signed, where the PRF secret key is stored along with the signing key.

DHE misconfiguration. Finally, our survey shows that there is an industry-wide configuration problem with the deployment of DHE key exchange. While 79% of web sites moved to RSA-2048 (compared to 6.14% in 2007), the vast majority of sites who use DHE set their Diffie-Hellman prime to 1024 bits. As a result, recovering one computed session key by a brute-force cryptanalytic attack requires breaking a 1024-bit Diffie-Hellman problem, not 2048-bit RSA. By 2014 the CA/Browser Forum will regard 1024-bit security as inadequate. We recommend that whenever possible, sites abandon DHE in favor of ECDHE using an elliptic curve with (presumed) security comparable to RSA-2048.

One could argue that 1024-bit Diffie-Hellman parameters are justified in this context because the Diffie-Hellman values are ephemeral and only used for a small number of sessions. Therefore, attacking a specific 1024-bit Diffie-Hellman problem will only expose a small number of sessions. The difficulty with this argument is that, if for whatever reason an attacker decides to target a particular high-value session, that session only enjoys 1024-bit security. In other words, even though only one session may be broken, that single session may be all the attacker needs.

VI. Conclusion

While the need for TLS forward secrecy has become more widely discussed over the recent years, it is critical that servers are configured and implemented correctly, and not otherwise, achieving a false sense of security. In this paper, we first investigate the deployment of various cryptographic algorithms of 473,802 TLS sites and reported that the majority of DHE-enabled sites are configured with weak DH parameters. We ran two performance experiments to evaluate various cipher suites that support forward secrecy, and point out that forward secrecy using elliptic curve cryptography is actually free in the face of traditional RSA algorithms. Lastly, we analyze client-side latencies in the wild measured from our ad experiment, and observe that ECC-based forward secrecy is also free on the client-side (although some Windows clients may experience a one-time delay for downloading the root certificate). To conclude, we recommend websites to move away from the RSA key exchange to provide forward secrecy, and deploy the ECDHE key exchange over the much slower DHE.

Acknowledgments

We thank Rick Andrews, Kaspar Brand and Ivan Ristic for providing feedback on drafts of the paper. This work was supported by NSF and a grant from Symantec.

References

[1] Codenomicon, “The HeartBleed bug,” http://heartbleed.com/, 2014.

[2] J. Ball, “NSA’s Prism surveillance program: how it works and what it can do,” http://www.theguardian.com/world/2013/jun/08/nsa-prism-server-collection-facebook-google, 2013.

[3] A. Langley, “Forward secrecy for Google HTTPS,” https://www.imperialviolet.org/2011/11/22/forwardsecret.html, 2011.

[4] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008.

[5] A. O. Freier, P. Karlton, and P. C. Kocher, “The Secure Sockets Layer (SSL) Protocol Version 3.0,” RFC 6101 (Historic), Internet Engineering Task Force, Aug. 2011.

[6] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS),” RFC 4492 (Informational), Internet Engineering Task Force, May 2006, updated by RFC 5246. [Online]. Available: http://www.ietf.org/rfc/rfc4492.txt

[7] E. Rescorla, “Security holes... who cares?” in *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[8] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, “When private keys are public: results from the 2008 Debian OpenSSL vulnerability,” in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, 2009.

[9] H. K. Lee, T. Malkin, and E. Nahum, “Cryptographic strength of SSL/TLS servers: current and recent practices,” in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, 2007.

[10] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, “Mining your Ps and Qs: Detection of widespread weak keys in network devices,” in *Proceedings of the 21st USENIX Security Symposium*, Aug. 2012.

[11] Qualys SSL Labs, “Trustworthy Internet Movement - SSL Pulse,” https://www.trustworthyinternet.org/ssl-pulse/.

[12] Electronic Frontier Foundation, “The EFF SSL Observatory,” https://www.eff.org/observatory.

[13] R. Holz, L. Braun, N. Kammenhuber, and G. Carle, “The SSL landscape: a thorough analysis of the x.509 PKI using active and passive measurements,” in *Proceedings of the 11th ACM SIGCOMM Conference on Internet Measurement*, 2011.

[14] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer, “Here’s my cert, so trust me, maybe?: understanding TLS errors on the web,” in *Proceedings of the 22nd international conference on World Wide Web*, 2013.

[15] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, “Analysis of the HTTPS certificate ecosystem,” in *Proceedings of the 13th ACM SIGCOMM Conference on Internet Measurement*, 2013.

[16] G. Apostolopoulos, V. Peris, and D. Saha, “Transport layer security: How much does it really cost?” in *Proceedings of the IEEE INFOCOM*, 1999.

[17] C. Coarfa, P. Druschel, and D. S. Wallach, “Performance analysis of TLS Web servers,” *ACM Trans. Comput. Syst.*, vol. 24, no. 1, pp. 39–69, Feb. 2006.

[18] V. Gupta, D. Stebila, S. Fung, S. C. Shantz, N. Gura, and H. Eberle, “Speeding up secure web transactions using elliptic curve cryptography,” in *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[19] V. Bernat, "SSL/TLS & Perfect Forward Secrecy," http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html, 2011.

[20] N. Mavrogiannopoulos, "The price to pay for perfect-forward secrecy," http://nmav.gnutls.org/2011/12/price-to-pay-for-perfect-forward.html, 2011.

[21] "Alexa Top 1,000,000 Sites," http://www.alexa.com/topsites.

[22] I. Ventura-Whiting, "SSLScan - Fast SSL Scanner," http://sourceforge.net/projects/sslscan/.

[23] The OpenSSL Project, "OpenSSL: The open source toolkit for SSL/TLS," http://www.openssl.org.

[24] T. L. MEUR, "Bug 319901 - missing ec and ecparam commands in openssl package," https://bugzilla.redhat.com/show_bug.cgi?id=319901.

[25] CA/Browser Forum, "Baseline requirements for the issuance and management of publicly-trusted certificates, v.1.0," https://www.cabforum.org/Baseline_Requirements_V1.pdf, 2011.

[26] A. Langley, "How to botch TLS forward secrecy," https://www.imperialviolet.org/2013/06/27/botchingpfs.html.

[27] I. Ristic, "Increasing DHE strength on Apache 2.4.x," http://blog.ivanristic.com/2013/08/increasing-dhe-strength-on-apache.html.

[28] E. Abalea, "Bug 49559 - Patch to add user-specified Diffie-Hellman parameters," https://issues.apache.org/bugzilla/show_bug.cgi?id=49559.

[29] K. Bhargavan, "Re: OpenSSL client DH group limits," http://marc.info/?l=openssl-dev&m=138373503327957.

[30] R. Graham, "Tor is still DHE 1024 (NSA crackable)," http://blog.erratasec.com/2013/09/tor-is-still-dhe-1024-nsa-crackable.html, 2013.

[31] Oracle, "JDK-6521495 : Lift 1024-bit long prime restriction on Diffie-Hellman," https://bugs.openjdk.java.net/browse/JDK-6521495.

[32] NIST, "Digital Signature Standard," in *Federal Information Processing Standards Publications, FIPS PUB 186-2*, 2000.

[33] E. Käsper, "Fast elliptic curve cryptography in OpenSSL," in *Proceedings of the international conference on Financial Cryptography and Data Security*, 2011.

[34] K. Brand, "[Apache-SVN] revision 1527295," http://svn.apache.org/viewvc?view=revision&revision=1527295.

[35] The Apache Software Foundation, "ab - Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html.

[36] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from DNS rebinding attacks," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[37] L.-S. Huang, E. Y. Chen, A. Barth, E. Rescorla, and C. Jackson, "Talking to yourself for fun and profit," in *Proceedings of the Web 2.0 Security and Privacy*, 2011.

[38] Z. Wang, "Navigation timing," http://www.w3.org/TR/navigation-timing/, 2012.

[39] Microsoft Support, "Windows root certificate program members - how windows updates root certificates," http://support.microsoft.com/kb/931125#MT5.

[40] P. Nguyen and I. Shparlinski, "The insecurity of the elliptic curve digital signature algorithm with partially known nonces," *Design Codes Cryptography*, vol. 30, no. 2, pp. 201–217, 2003.

[41] T. Duong and J. Rizzo, "BEAST," http://vnhacker.blogspot.com/2011/09/beast.html, 2011.

[42] N. J. Al Fardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in *IEEE Symposium on Security and Privacy*, 2013.

Appendix

A. SSL/TLS protocol versions

Table X presents the percentage of unique hosts that support each SSL/TLS protocol version (out of 473,802 unique hosts that completed our TLS scan). Each unique host may support multiple protocol versions. In comparison to Lee et al.'s survey in November 2006 [9], the most significant difference is that SSLv2 support has dramatically reduced from 85% down to 22%, while TLSv1.1 and TLSv1.2 have slowly gained some adoption. Our results of TLS adoption roughly match a recent survey by SSL Pulse as of August 2013 [11].

TABLE X: Protocol version support

Version	Hosts	SSL Pulse[a]	IMC'07
SSLv2	105,239 (22.2%)	26.9%	85.37%
SSLv3	470,409 (99.2%)	99.7%	97.92%
TLSv1	471,458 (99.5%)	99.3%	98.36%
TLSv1.1	79,890 (16.8%)	15.4%	
TLSv1.2	84,406 (17.8%)	17.8%	

[a] SSL Labs [11] surveyed 168,088 sites as of August 2013.

B. Encryption methods

Table XI compares the percentage of unique hosts that support each encryption method and key size. We show, in a separate column, the percentage of including all protocol versions, and the percentage for TLSv1.2 specifically. Overall, we observe that 3DES is currently the most supported cipher. Both RC4-128 and AES-128 have more than 90% adoption. Interestingly, 256-bit AES has no less adoption rate than 128-bit AES. Comparing to the survey in 2006 [9], we observe a dramatic increase in AES-128 adoption from 2.05% to 90.3%. Also, support for DES and RC2 have dropped significantly.

TABLE XI: Encryption method support

Method	Hosts	TLSv1.2[a]	IMC'07
3DES-168	446,119 (94.1%)	81,085 (96.0%)	97.50%
RC4-128	436,994 (92.2%)	77,271 (91.5%)	98.58%
AES-256	428,176 (90.3%)	82,191 (97.3%)	56.37%
AES-128	428,011 (90.3%)	82,420 (97.6%)	2.05%
DES-56	153,243 (32.3%)	10,400 (12.3%)	62.29%
Camellia-128	150,421 (31.7%)	26,656 (31.5%)	
Camellia-256	150,443 (31.7%)	26,637 (31.5%)	
RC4-40	127,209 (26.8%)	7,725 (9.1%)	91.75%
RC2-40	124,890 (26.3%)	7,677 (9.0%)	90.31%
DES-40	114,036 (24.0%)	7,715 (9.1%)	66.55%
SEED-128	85,272 (17.9%)	16,837 (19.9%)	
RC2-128	81,570 (17.2%)	0 (0.0%)	83.78%
AES-GCM-128	56,098 (11.8%)	56,098 (66.5%)	
AES-GCM-256	55,614 (11.7%)	55,614 (66.5%)	
IDEA-128	37,735 (7.9%)	8,807 (10.4%)	
Null	443 (0.0%)	30 (0.0%)	

[a] This column shows results for only TLSv1.2.

Unfortunately, almost a third of all TLS sites still support DES-56, which is considered insecure. Roughly a quarter of all TLS sites support 40-bit encryption, including RC4-40, RC2-40 and DES-40. We even found 443 sites supporting null ciphers. Lastly, we observe that the new AES Galois Counter Mode (AES-GCM) cipher is enabled on 66.5% of the TLSv1.2 sites. AES-GCM is immune to BEAST [41] and the Lucky 13 timing attack [42].

C. Message authentication methods

Table XII shows the website support of different message authentication methods. SHA-1 is pervasively (99.9%) supported, while MD5 has dropped from 99.83% to 72.3% since 2006 [9]. Here, the Authenticated Encryption with Associated Data (AEAD) method means that the message is encrypted and authenticated using a single key (as for AES-GCM) rather than a separate HMAC.

TABLE XII: Message authentication method support

Method	Hosts	IMC'07
SHA-1	473,462 (99.9%)	99.47%
MD5	342,618 (72.3%)	99.83%
SHA-256	69,266 (14.6%)	
AEAD	56,200 (11.8%)	
SHA-384	40,915 (8.6%)	